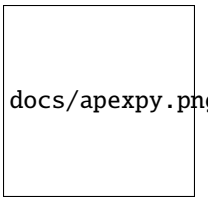
ApexPy

Release "2.0.1"

van der Meeren, Christer and Laundal, Karl M. and Burrell, Angeli

Apr 11, 2023

CONTENTS


docs/apexpy.png

CHAPTER

ONE

OVERVIEW

This is a Python wrapper for the Apex fortran library by Emmert et al. [2010][1], which allows converting between geodetic, modified apex, and quasi-dipole coordinates as well as getting modified apex and quasi-dipole base vectors (Richmond [1995][2]). The geodetic system used here is WGS84. MLT calculations are also included. The package is free software (MIT license).

1.1 Quick start

Install from PyPI using `pip`:

```
pip install apexpy
```

This assumes that the same version of libgfortran is installed in the same location as when the pip wheel was built (if a wheel was used). If not, you may have trouble importing apexpy. If you run into trouble, try the command:

```
pip install --no-binary :apexpy: apexpy
```

which requires both libgfortran and gfortran to be installed on your system. More detailed installation instructions (and troubleshooting) is available in the documentation.

Conversion is done by creating an `Apex` object and using its methods to perform the desired calculations. Some simple examples:

```
from apexpy import Apex
import datetime as dt
atime = dt.datetime(2015, 2, 10, 18, 0, 0)
apex15 = Apex(date=2015.3)  # dt.date and dt.datetime objects also work

# Geodetic to apex, scalar input
mlat, mlon = apex15.convert(60, 15, 'geo', 'apex', height=300)
print("{:.12f}, {:.12f}".format(mlat, mlon))
57.477310180664, 93.590156555176

# Apex to geodetic, array input
glat, glon = apex15.convert([90, -90], 0, 'apex', 'geo', height=0)
```

(continues on next page)

[1] Emmert, J. T., A. D. Richmond, and D. P. Drob (2010), A computationally compact representation of Magnetic-Apex and Quasi-Dipole coordinates with smooth base vectors, J. Geophys. Res., 115(A8), A08322, doi:10.1029/2010JA015326.

[2] Richmond, A. D. (1995), Ionospheric Electrodynamics Using Magnetic Apex Coordinates, Journal of geomagnetism and geoelectricity, 47(2), 191–212, doi:10.5636/jgg.47.191.

(continued from previous page)

```
print(["{:.12f}, {:.12f}".format(ll, glon[i]) for i,ll in enumerate(glat)])
['83.103820800781, -84.526657104492', '-74.388252258301, 125.736274719238']

# Geodetic to magnetic local time
mlat, mlt = apex15.convert(60, 15, 'geo', 'mlt', datetime=atime)
print("{:.12f}, {:.12f}".format(mlat, mlt))
56.598316192627, 19.107861709595

# can also convert magnetic longitude to mlt
mlt = apex15.mlon2mlt(120, atime)
print("{:.2f}".format(mlt))
20.90
```

If you don't know or use Python, you can also use the command line. See details in the full documentation (link in the section below).

1.2 Documentation

https://apexpy.readthedocs.io/en/latest

1.3 References

1.4 Badges

docs	
tests	
package	

CHAPTER

TWO

INSTALLATION

2.1 Standard Installation

The recommended (and most straightforward) method for users to install *apexpy* is through PyPI. From the command line use `pip`[1]:

```
pip install apexpy
```

You should be able to import *apexpy* and run basic conversions as shown in the examples. If you get errors or warnings upon importing, see below for more advanced options and troubleshooting.

2.2 Tested environments

The package has been tested with the following setups (others might work, too):

- Windows (64 bit Python), Linux (64 bit), and Mac (64 bit)
- Python 3.7, 3.8, 3.9, 3.10

2.3 Advanced Installation

If you cannot install *apexpy* from the distribtuion available on PyPI, you will have to use one of the following more advanced options. These are generally only recommended if you are planning on developing or modifying the *apexpy* source code.

The code behind this package is written in Fortran. Because of this, you **MUST** have a fortran compiler installed on your system before you attempt the following steps. Gfortran is a free compiler that can be installed, if one is not already available on your system. If you are installing this or MinGW in Windows, make sure you install it **after** installing the Windows Microsoft C++ Build tools. You must also make sure that the compilers and Python that are installed both use the same processing standard (either 32-bit or 64-bit). The *apexpy* installation has been tested successfully with gfortran 7 and some more recent versions. Earlier versions of gfortran may not work and are not recommended.

Installation also requires a C compiler of the same type as the fortran compiler. GCC is a free compiler that works with Gfortran and can be installed from a variety of sources and standard package managers. It is recommended that you check to see if you have gcc available on your system before installing as it is relatively common and multiple competing versions may cause problems if paths are not managed carefully.

This package requires NumPy, which you can install alone or as a part of SciPy. Some Python distributions come with NumPy/SciPy pre-installed. For Python distributions without NumPy/SciPy, various package managers for different

[1] pip is included with Python 2 from v2.7.9 and Python 3 from v3.4. If you don't have pip, get it here.

operating systems allow for simple local installation (as directed on the SciPy installation page. Pip should install NumPy automatically when installing *apexpy*, but if not, install it manually before attempting to install *apexpy*. *apexpy* is not compatible with NumPy version 1.19.X, older and newer versions of NumPy work without issue.

IMPORTANT: If you are struggling with installing *apexpy* and trying some of the following options, it is recommended that you user the `--no-cache` flag with pip to avoid repeatedly reinstalling the same non-functional build.

2.3.1 Install from GitHub

apexpy can be installed from the source code on GitHub, so long as a fortran compiler is available:

```
pip install git+https://github.com/aburrell/apexpy.git
```

This is advantageous if you would like to install from a particular branch or tag instead of the latest published stable release on PyPI. Do this by appending *@target-branch* to the end of the above command. For instance, if you would like to install from the develop branch instead of main, the appropriate command would be:

```
pip install git+https://github.com/aburrell/apexpy.git@develop
```

2.3.2 Install without Wheels

Many times, skipping building wheels locally will solve installation problems, but it requires that both libgfortran and gfortran are installed on your system:

```
pip install --no-binary :apexpy: apexpy
```

This is the default option for Linux, and so should not be an issue there. On Windows with the Mingw32 compiler, you might find this information useful for helping build *apexpy*.

2.3.3 Install against an incompatible numpy version

pip install apexpy –no-build-isolation –no-cache

2.3.4 Installation using CI Wheels

If your local set up is essentially identical to one of the CI test environments, then you can use one of the wheel artifacts to install *apexpy*. The list of artifacts may be found here.

To download an artifact:

1. If you don't have a GitHub Personal Access Token, follow these instructions to create one.
2. Run `curl -v -H "Authorization: token <GITHUB-ACCESS-TOKEN>" https://api.github.com/repos/aburrell/apexpy/actions/artifacts/<ARTIFACT-ID>/zip`, where <ITEM> should be replaced with the appropriate item string.
3. Copy the URL from the `Location` output produced by the previous command into a browser, which will download a zip archive into your standard download location. Alternatively (or if this doesn't work) you can use *wget* to retrieve the archive.
4. Copy the zip archive into the `apexpy/dist` directory and unzip.
5. Check the archive for the expected matrix of `*.whl` objects

To install, use `pip install .`

2.3.5 Build from Source

If you intend to modify or contribute to *apexpy*, you should install *apexpy* by forking the repository and installing it locally or within a virtual environment. After cloning the fork (see *Contributing*), you may install by:

```
cd apexpy
python -m build .
pip install .
```

Note that the `-e` flag for pip, which performs what used to be `python setup.py develop`, isn't used here. That's because meson currently doesn't support develop style builds.

If the above command doesn't work for you (as may be the case for Windows), you can try:

```
cd apexpy
meson setup build
ninja -j 2 -C build
cd build
meson install
```

2.3.6 Specifying Compilers

When you install *apexpy* from the command line you can specify the compilers you would like to use. These can be changed by altering the `CC` and `FC` environment variables on your computer:

```
FC=/path/to/correct/gfortran CC=/path/to/correct/gcc python -m build
pip install .
```

This can be useful your system has multiple versions of gfortran or gcc and the default is not appropriate (ie., an older version). If using an Intel compiler, you will need to clone the repository locally and uncomment a line at the top of `src/fortranapex/igrf.f90` to ensure all necessary libraries are imported.

2.3.7 When All Else Fails

Because the base code is in Fortran, installation can be tricky and different problems can arise even if you already have a compiler installed. The following are a series of installation commands that users have reported working for different system configurations. We have not been able to reproduce some of the issues users report and cannot fully explain why some of the options work, none the less they are recorded here as they may be useful to other users. If you feel like you can provide more insight on the situations where these commands are appropriate or discover a new installation process that works for your system when none of the previously described standard approaches work, please consider contributing to this documentation (see *Contributing*).

Problems have been encountered when installing in a conda environment. In this case, pip seems to ignore the installed NumPy version when installing. This appears to result in a successful installation that fails upon import or causes a RuntimeError. This happens when the version of NumPy used to build *apexpy* is newer than the system version of NumPy (NumPy may not be forwards compatible). In this case, try:

```
pip install apexpy --no-build-isolation --no-cache
```

Apple Silicon systems require certain compilation flags to deal with memory problems. *apexpy* may appear to install and import correctly, but then fail with BUS errors when used. In this case, the following command has worked:

```
CFLAGS="-falign-functions=8 ${CFLAGS}" pip install --no-binary :apexpy: apexpy
```

If you are on Apple and encounter a library error such as `ld: library not found for -lm`, you will need to provide an additional linking flag to the Mac OSX SDK library:

```
LDFLAGS="-L/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/lib ${LDFLAGS}" pip␣
↪install .
```

This example assumes you are building locally from the cloned Git repository. Issues on Mac OS have also been encountered when using clang for `CC` alongside gfortran. This resulted in a seemly successful installation with *apexpy* reporting that fortranapex cannot be imported.

Some users have reported unusual behavior when using Anaconda on Apple Silicon systems. Anaconda will attempt to build and install the Intel versions of wheels instead of the M1 versions and run everything through Rosetta. This configuration has not been fully evaluated, but it results in a seemly successful installation with *apexpy* reporting that fortranapex cannot be imported. Users should confirm that wheels created by conda (both for apexpy and other packages) end in `arm64.whl` not `osx-64.whl`. If the later is true, users should consider uninstalling anaconda completely, and instead installing miniconda following these instructions, which has been confirmed to work. **WARNING:** This will remove any environments you have set up and likely undo all IDE settings, so be cautious and consider backing up your work first!

Windows systems are known to have issues with Fortran-based codes. The Windows testing we do uses miniconda, so we recommend using the Anaconda environment. One problem that has been encountered is a lack of LAPACK/BLAS tools that causes NumPy to not behave as expected. This can be fixed by installing scipy before NumPy and then installing *apexpy*.

CHAPTER

THREE

EXAMPLES

Here are some simple examples that will show how to use some of the Apex methods and support functions.

3.1 Using apexpy

The *Apex* class contains most of the methods you'll want to use when converting between geodetic and apex or quasi-dipole coordinates. The *convert()* method is designed to be the primary user interface for coordinate conversion. For full documentation of this and other class methods, see the reference for *Apex*. Some simple examples to get you going follow below.

3.1.1 Initialize the Apex class

The Apex class requires a date and time as well as a reference height when initialized. If you don't supply one, then the class will default to the current time (in Universal Time) and a reference height of 0 km. For this example, we will specify the time. Time can be supplied as either a decimal year, `datetime` object, or a `date` object.

```
import apexpy
apex_out = apexpy.Apex(date=2015.3)
print(apex_out)
```

This yields:

```
Apex class conversions performed with
-------------------------------------
Decimal year: 2015.30000000
Reference height: 0.000 km
Earth radius: 6371.009 km

Coefficient file: '/path/to/programs/apexpy/src/apexpy/apexsh.dat'
Cython Fortran library: '/path/to/programs/Git/apexpy/src/apexpy/fortranapex.cpython-37m-
↪darwin.so'
```

3.1.2 Convert from Geodetic to Magnetic Coordinates

You can use the initialized *Apex()* object to convert from geodetic coordinates to magnetic coordinates and back again. When converting to and from apex coordinates, you need to supply the height of the observations.

```
alat, alon = apex_out.convert(60, 15, 'geo', 'apex', height=300)
print("{:.12f}, {:.12f}".format(alat, alon))
57.477310180664, 93.590156555176
```

For quasi-dipole coordinates, this isn't necessary.

```
qlat, qlon = apex_out.convert(60, 15, 'geo', 'qd')
print("{:.12f}, {:.12f}".format(qlat, qlon))
56.598316192627, 93.174751281738
```

You can calculate multiple locaitons at once using arrays, as long as the inputs are broadcastable. For example, you can provide a list or array of different latitudes for a single longitude (and height). It is also acceptable to provide a list or array of the same shape that provides paired latitude, longitude, and heights (if needed). However, you can't provide mismatched array or list inputs. Here is an example where we convert from apex coordinates to geodetic coordinates for two different latitudes at the same longitude and height.

```
glat, glon = apex_out.convert([90, -90], 0, 'apex', 'geo', height=0)
print(["{:.12f}, {:.12f}".format(ll, glon[i]) for i,ll in enumerate(glat)])
['83.103820800781, -84.526657104492', '-74.388252258301, 125.736274719238']
```

3.1.3 Convert to Magnetic Local Time

When converting to magnetic local time (MLT), the convert function requires a datetime input alongside a latitude and longitude.

```
import datetime as dt
utime = dt.datetime(2015, 2, 10, 18, 0, 0)
mlat, mlt = apex_out.convert(60, 15, 'geo', 'mlt', datetime=utime)
print("{:.12f}, {:.12f}".format(mlat, mlt))
56.598316192627, 19.107861709595
```

If you already have magnetic longitude, you can also calculate MLT using *mlon2mlt()*.

```
mlt = apex_out.mlon2mlt(120, utime)
print("{:.2f}".format(mlt))
20.90
```

3.2 Command-Line Interface

The Python package also installs a command called `apexpy` which allows using the *convert()* method from the command line. The command-line interface allows you to make use of the Python library even if you don't know or use Python. See the reference for *Command-line interface* for a list of arguments to the commands. Below are some simple usage examples.

3.2.1 Running Many Locations

You can convert many locations at a single time using an input file. To follow this example, create a file called `input.txt` with the input latitudes and longitudes on each row separated by whitespace as shown below.

```
# gdlat gdlon
# comment lines like these are ignored
60 15
61 15
62 15
```

To convert these geodetic coordinates to apex coordinates using the magnetic field model for the date 2015-01-01 using a height of 300 km, run the command `apexpy geo apex 20150101 --height 300 -i input.txt -o output.txt` (in this specific example you could also just use `2015` or `201501` for the date). This will create an output file named `output.txt` that looks like this:

```
57.46954727 93.63981628
58.52270126 94.04476166
59.57146454 94.47725677
```

3.2.2 Running One Location

If you don't have a lot of data to process, you can skip the input and output files using command-line piping. The `echo` command will provide the input geodetic latitude and longitude in this example and the output appears on the screen:

```
$ echo 60 15 | apexpy geo apex 2015 --height 300
57.46954727 93.63981628
```

3.2.3 MLT Conversions

The previous examples all showed how to convert from geodetic to apex coordinates, but you can convert to and from any of the coordinate systems supported by *convert()*. In this example, we show how to convert from geodetic latitude and longitude to apex latitude and magnetic local time (MLT).

MLT conversion works in much the same way as any other coordinate conversion, but requires both date and time (`YYYYMMDDHHMMSS`). For example, if you want to find the MLT (and the apex latitude) at geodetic coordinates (60, 15) for midnight on the day 2015-01-01, run `echo 60 15 | apexpy geo mlt 20150101000000`. The output will look like this:

```
56.59033585 1.03556417
```

3.3 Calculate L-Shells

L-shells are the apex heights of specified magnetic field lines in units of Earth radii where L=0 corresponds to the center of the Earth. You can calculate the L-shells seen by a given instrument using the *geo2apex()* and *get_apex()* methods. An example of this is shown for a single point along the orbit of the International Space Station (ISS) below.

```
import apexpy
import datetime as dt
```

(continues on next page)

(continued from previous page)

```
# Set the location of the ISS in geodetic coordinates at a single time
stime = dt.datetime(2021, 3, 15, 15, 6)
iss_lat = 9.8
iss_lon = 142.2
iss_alt = 419.0

# Get the apex lat
apex_iss = apexpy.Apex(stime, refh=iss_alt)
alat, alon = apex_iss.geo2apex(iss_lat, iss_lon, iss_alt)

# Get the apex height
aalt = apex_iss.get_apex(alat, iss_alt)

# Convert from apex height in km to L-shell
L_iss = 1.0 + aalt / apex_iss.RE
print("apex height={:.3f} km, L={:.2f}".format(aalt, L_iss))
apex height=428.007 km, L=1.07
```

3.4 Trace a Field Line

It can be useful to trace a field line with a specified apex height across a range of known latitudes. This can then be useful when plotting field lines at a particular magnetic meridian or used to gather data along the same field line, but measured by different instruments.

```
import numpy as np

# Continue form the previous example, define the field line at all latitudes
lats = np.linspace(-90, 90, 181)
alts = apex_iss.get_height(lats, aalt)

# Select the locations with positive altitudes (above the Earth's surface)
iline = np.where(alts >= 0.0)[0]

# Print the latitude limits for which the field line is above the surface
# of the Earth
print("lat={:.2f} deg, alt={:.2f} km; lat={:.2f} deg, alt={:.2f} km".format(
    lats[iline[0]], alts[iline[0]], lats[iline[-1]], alts[iline[-1]]))
lat=-14.00 deg, alt=30.09 km; lat=14.00 deg, alt=30.09 km
```

3.5 Convert from various non-Magnetic Coordinate Systems

Usually non-magnetic coordinates for space sciences are provided in geodetic (geographic) coordinates, where the latitude is the angle between the equatorial plane and the normal to the ellipsoid surface at the desired location. There are several different reference ellipsoids that may be used, but the most common (and the one used by *Apex* is WGS84, the World Geodetic System 1984 (as described in, for example, Snay and Soler[1]). However, it frequently makes sense for a particular instrument to use a different reference ellipsoid or another type of coordinate system.

[1] Snay and Soler (1999) Modern Terrestrial Reference Systems (Part 1), Professional Surveyor.

Different types of geodetic or geocentric coordinate conversions are not supported within apexpy, because they are not a magnetic coordinate transformations. We recommend first converting your other non-magnetic coordinates to geodetic WGS84 coordinates and then performing the desired conversion to apex or quasi-dipole coordinates.

3.5.1 Geocentric Example

One commonly encountered alternative to geodetic latitude is geocentric latitude. Geocentric latitude is the angle between the equatorial plane and a line joining the center of the Earth and the desired location. If you have data in geocentric coordinates you can convert them to geodetic using a simple equation layed out in equation 3-28 of Snyder[2].

```
import numpy as np

def gc_to_gd(gc_lat, e_sq):
    """Convert from geocentric to geodetic

    Parameters
    ----------
    gc_lat : array-like
       Geocentric latitude in degrees
    e_sq : float
        First eccentricity squared, unitless

    Returns
    -------
    gd_lat : array-like
        Geodetic latitude in degrees
    """
    gd_lat = np.arctan(np.tan(np.radians(gc_lat)) / (1.0 - e_sq))
    return np.degrees(gd_lat)
```

The function above requires the first eccentricity of the reference ellipsoid. This example uses the `pyproj` library[3] to get the WGS84 ellipsoid data, but the function shown will take any float. This lets you decide the level of precision you need in your coordinate transformation.

```
import pyproj

 # Initialize the WGS84 reference ellipsoid
 wgs84_geod = pyproj.crs.GeographicCRS(name='WGS84').get_geod()
 print("{:.12f}".format(wgs84_geod.es))
 0.006694379990
```

Now, try converting from geocentric to quasi-dipole coordinates. In this example you need to supply height, because the coordinate transformation from geodetic to quasi-dipole takes place by converting from geodetic to apex and then from apex to quasi-dipole.

```
import apexpy

# Define the starting values
year = 2015.3
gc_lat = 45.0
glon = 0.0
```

(continues on next page)

[2] Snyder, J. P. Map projections — A working manual. Professional Paper 1395, U.S. Geological Survey, 1987. doi:10.3133/pp1395.
[3] pyproj GitHub page.

(continued from previous page)

```
height = 0.0

# Get the quasi-dipole coordiantes
apex_out = apexpy.Apex(date=year)
qlat, qlon = apex_out.geo2qd(gc_to_gd(gc_lat, wgs84_geod.es), glon, height)
print("{:.12f}, {:.12f}".format(qlat, qlon))
39.852313995361, 76.711242675781
```

CHAPTER

FOUR

REFERENCE

The *apexpy.Apex* class is used for all the main functionality (converting between coordinate systems, field line mapping, and calculating base vectors). The *apexpy.helpers* sub-module includes additional functions that may be useful, especially *subsol()*. The `apexpy.fortranapex` module is the interface to the apex Fortran library by Emmert et al. [2010][1]. The interface is not documented. Use *apexpy.Apex* for all conversions and calculations. You can find some documentation of the actual Fortran library in the source file apexsh.f90. These functions may also be accessed through the command-line interface.

4.1 API

4.1.1 apexpy

Submodules

apexpy.__main__

Entry point for the Command Line Interface

Module Contents

Functions

main()	Entry point for the script

Attributes

STDIN
STDOUT

apexpy.__main__.**STDIN**

[1] Emmert, J. T., A. D. Richmond, and D. P. Drob (2010), A computationally compact representation of Magnetic-Apex and Quasi-Dipole coordinates with smooth base vectors, J. Geophys. Res., 115(A8), A08322, doi:10.1029/2010JA015326.

apexpy.__main__.**STDOUT**

apexpy.__main__.**main**()

Entry point for the script

apexpy._gcc_build_bitness

Detect bitness (32 or 64) of Mingw-w64 gcc build target on Windows.

From SciPy v1.10.0.dev0

Module Contents

Functions

main()	

apexpy._gcc_build_bitness.**main**()

apexpy.apex

Classes that make up the core of apexpy.

Module Contents

Classes

Apex	Calculates coordinate conversions, field-line mapping, and base vectors.

exception apexpy.apex.**ApexHeightError**

Bases: ValueError

Specialized ValueError definition, to be used when apex height is wrong.

class apexpy.apex.**Apex**(*date=None*, *refh=0*, *datafile=None*, *fortranlib=None*)

Bases: object

Calculates coordinate conversions, field-line mapping, and base vectors.

Parameters

- **date** (NoneType, float, dt.date, or dt.datetime, optional) – Determines which IGRF coefficients are used in conversions. Uses current date as default. If float, use decimal year. If None, uses current UTC. (default=None)
- **refh** (*float, optional*) – Reference height in km for apex coordinates, the field lines are mapped to this height. (default=0)

- **datafile** (*str or NoneType, optional*) – Path to custom coefficient file, if None uses *apexsh.dat* file (default=None)
- **fortranlib** (*str or NoneType, optional*) – Path to Fortran Apex CPython library, if None uses linked library file (default=None)

Variables

- **year** (*float*) – Decimal year used for the IGRF model
- **RE** (*float*) – Earth radius in km, defaults to mean Earth radius
- **refh** (*float*) – Reference height in km for apex coordinates
- **datafile** (*str*) – Path to coefficient file
- **fortranlib** (*str*) – Path to Fortran Apex CPython library
- **igrf_fn** (*str*) – IGRF coefficient filename

Notes

The calculations use IGRF-13 with coefficients from 1900 to 2025[1].

The geodetic reference ellipsoid is WGS84.

References

__repr__()

Produce an evaluatable representation of the Apex class.

__str__()

Produce a user-friendly representation of the Apex class.

__eq__(*comp_obj*)

Performs equivalency evaluation.

Parameters
comp_obj – Object of any time to be compared to the class object

Returns
bool or NotImplemented – True if self and comp_obj are identical, False if they are not, and NotImplemented if the classes are not the same

_apex2qd_nonvectorized(*alat*, *alon*, *height*)

Convert from apex to quasi-dipole (not-vectorised)

Parameters

- **alat** ((*float*)) – Apex latitude in degrees
- **alon** ((*float*)) – Apex longitude in degrees
- **height** ((*float*)) – Height in km

Returns

- **qlat** ((*float*)) – Quasi-dipole latitude in degrees
- **qlon** ((*float*)) – Quasi-diplole longitude in degrees

[1] Thébault, E. et al. (2015), International Geomagnetic Reference Field: the 12th generation, Earth, Planets and Space, 67(1), 79, doi:10.1186/s40623-015-0228-9.

`_qd2apex_nonvectorized`(*qlat*, *qlon*, *height*)

Converts quasi-dipole to modified apex coordinates.

Parameters

- **qlat** (*float*) – Quasi-dipole latitude
- **qlon** (*float*) – Quasi-dipole longitude
- **height** (*float*) – Altitude in km

Returns

- **alat** (*float*) – Modified apex latitude
- **alon** (*float*) – Modified apex longitude

Raises

ApexHeightError – if apex height < reference height

`_map_EV_to_height`(*alat*, *alon*, *height*, *newheight*, *data*, *ev_flag*)

Maps electric field related values to a desired height

Parameters

- **alat** (*array-like*) – Apex latitude in degrees.
- **alon** (*array-like*) – Apex longitude in degrees.
- **height** (*array-like*) – Current altitude in km.
- **new_height** (*array-like*) – Desired altitude to which EV values will be mapped in km.
- **data** (*array-like*) – 3D value(s) for the electric field or electric drift
- **ev_flag** (*str*) – Specify if value is an electric field ('E') or electric drift ('V')

Returns

data_mapped (*array-like*) – Data mapped along the magnetic field from the old height to the new height.

Raises

`ValueError` – If an unknown *ev_flag* or badly shaped data input is supplied.

`_get_babs_nonvectorized`(*glat*, *glon*, *height*)

Get the absolute value of the B-field in Tesla

Parameters

- **glat** (*float*) – Geodetic latitude in degrees
- **glon** (*float*) – Geodetic longitude in degrees
- **height** (*float*) – Altitude in km

Returns

babs (*float*) – Absolute value of the magnetic field in Tesla

`convert`(*lat*, *lon*, *source*, *dest*, *height=0*, *datetime=None*, *precision=1e-10*, *ssheight=50 * 6371*)

Converts between geodetic, modified apex, quasi-dipole and MLT.

Parameters

- **lat** (*array_like*) – Latitude in degrees
- **lon** (*array_like*) – Longitude in degrees or MLT in hours

- **source** (*str*) – Input coordinate system, accepts 'geo', 'apex', 'qd', or 'mlt'
- **dest** (*str*) – Output coordinate system, accepts 'geo', 'apex', 'qd', or 'mlt'
- **height** (*array_like, optional*) – Altitude in km
- **datetime** (`datetime.datetime`) – Date and time for MLT conversions (required for MLT conversions)
- **precision** (*float, optional*) – Precision of output (degrees) when converting to geo. A negative value of this argument produces a low-precision calculation of geodetic lat/lon based only on their spherical harmonic representation. A positive value causes the underlying Fortran routine to iterate until feeding the output geo lat/lon into geo2qd (APXG2Q) reproduces the input QD lat/lon to within the specified precision (all coordinates being converted to geo are converted to QD first and passed through APXG2Q).
- **ssheight** (*float, optional*) – Altitude in km to use for converting the subsolar point from geographic to magnetic coordinates. A high altitude is used to ensure the subsolar point is mapped to high latitudes, which prevents the South-Atlantic Anomaly (SAA) from influencing the MLT.

Returns

- **lat** (*ndarray or float*) – Converted latitude (if converting to MLT, output latitude is apex)
- **lon** (*ndarray or float*) – Converted longitude or MLT

Raises

`ValueError` – For unknown source or destination coordinate system or a missing or badly formed latitude or datetime input

`geo2apex`(*glat*, *glon*, *height*)

Converts geodetic to modified apex coordinates.

Parameters

- **glat** (*array_like*) – Geodetic latitude
- **glon** (*array_like*) – Geodetic longitude
- **height** (*array_like*) – Altitude in km

Returns

- **alat** (*ndarray or float*) – Modified apex latitude
- **alon** (*ndarray or float*) – Modified apex longitude

`apex2geo`(*alat*, *alon*, *height*, *precision=1e-10*)

Converts modified apex to geodetic coordinates.

Parameters

- **alat** (*array_like*) – Modified apex latitude
- **alon** (*array_like*) – Modified apex longitude
- **height** (*array_like*) – Altitude in km
- **precision** (*float, optional*) – Precision of output (degrees). A negative value of this argument produces a low-precision calculation of geodetic lat/lon based only on their spherical harmonic representation. A positive value causes the underlying Fortran routine to iterate until feeding the output geo lat/lon into geo2qd (APXG2Q) reproduces the input QD lat/lon to within the specified precision.

Returns

- **glat** (*ndarray or float*) – Geodetic latitude
- **glon** (*ndarray or float*) – Geodetic longitude
- **error** (*ndarray or float*) – The angular difference (degrees) between the input QD coordinates and the qlat/qlon produced by feeding the output glat and glon into geo2qd (APXG2Q)

`geo2qd`(*glat*, *glon*, *height*)

Converts geodetic to quasi-dipole coordinates.

Parameters

- **glat** (*array_like*) – Geodetic latitude
- **glon** (*array_like*) – Geodetic longitude
- **height** (*array_like*) – Altitude in km

Returns

- **qlat** (*ndarray or float*) – Quasi-dipole latitude
- **qlon** (*ndarray or float*) – Quasi-dipole longitude

`qd2geo`(*qlat*, *qlon*, *height*, *precision=1e-10*)

Converts quasi-dipole to geodetic coordinates.

Parameters

- **qlat** (*array_like*) – Quasi-dipole latitude
- **qlon** (*array_like*) – Quasi-dipole longitude
- **height** (*array_like*) – Altitude in km
- **precision** (*float, optional*) – Precision of output (degrees). A negative value of this argument produces a low-precision calculation of geodetic lat/lon based only on their spherical harmonic representation. A positive value causes the underlying Fortran routine to iterate until feeding the output geo lat/lon into geo2qd (APXG2Q) reproduces the input QD lat/lon to within the specified precision.

Returns

- **glat** (*ndarray or float*) – Geodetic latitude
- **glon** (*ndarray or float*) – Geodetic longitude
- **error** (*ndarray or float*) – The angular difference (degrees) between the input QD coordinates and the qlat/qlon produced by feeding the output glat and glon into geo2qd (APXG2Q)

`apex2qd`(*alat*, *alon*, *height*)

Converts modified apex to quasi-dipole coordinates.

Parameters

- **alat** (*array_like*) – Modified apex latitude
- **alon** (*array_like*) – Modified apex longitude
- **height** (*array_like*) – Altitude in km

Returns

- **qlat** (*ndarray or float*) – Quasi-dipole latitude
- **qlon** (*ndarray or float*) – Quasi-dipole longitude

Raises
ApexHeightError – if *height* > apex height

`qd2apex`(*qlat*, *qlon*, *height*)

Converts quasi-dipole to modified apex coordinates.

Parameters

- **qlat** (*array_like*) – Quasi-dipole latitude
- **qlon** (*array_like*) – Quasi-dipole longitude
- **height** (*array_like*) – Altitude in km

Returns

- **alat** (*ndarray or float*) – Modified apex latitude
- **alon** (*ndarray or float*) – Modified apex longitude

Raises
ApexHeightError – if apex height < reference height

`mlon2mlt`(*mlon*, *dtime*, *ssheight=318550*)

Computes the magnetic local time at the specified magnetic longitude and UT.

Parameters

- **mlon** (*array_like*) – Magnetic longitude (apex and quasi-dipole longitude are always equal)
- **dtime** (`datetime.datetime`) – Date and time
- **ssheight** (*float, optional*) – Altitude in km to use for converting the subsolar point from geographic to magnetic coordinates. A high altitude is used to ensure the subsolar point is mapped to high latitudes, which prevents the South-Atlantic Anomaly (SAA) from influencing the MLT. The current default is 50 * 6371, roughly 50 RE. (default=318550)

Returns
mlt (*ndarray or float*) – Magnetic local time in hours [0, 24)

Notes

To compute the MLT, we find the apex longitude of the subsolar point at the given time. Then the MLT of the given point will be computed from the separation in magnetic longitude from this point (1 hour = 15 degrees).

`mlt2mlon`(*mlt*, *dtime*, *ssheight=318550*)

Computes the magnetic longitude at the specified MLT and UT.

Parameters

- **mlt** (*array_like*) – Magnetic local time
- **dtime** (`datetime.datetime`) – Date and time
- **ssheight** (*float, optional*) – Altitude in km to use for converting the subsolar point from geographic to magnetic coordinates. A high altitude is used to ensure the subsolar point is mapped to high latitudes, which prevents the South-Atlantic Anomaly (SAA) from influencing the MLT. The current default is 50 * 6371, roughly 50 RE. (default=318550)

Returns

mlon (*ndarray or float*) – Magnetic longitude [0, 360) (apex and quasi-dipole longitude are always equal)

Notes

To compute the magnetic longitude, we find the apex longitude of the subsolar point at the given time. Then the magnetic longitude of the given point will be computed from the separation in magnetic local time from this point (1 hour = 15 degrees).

`map_to_height`(*glat*, *glon*, *height*, *newheight*, *conjugate=False*, *precision=1e-10*)

Performs mapping of points along the magnetic field to the closest or conjugate hemisphere.

Parameters

- **glat** (*array_like*) – Geodetic latitude
- **glon** (*array_like*) – Geodetic longitude
- **height** (*array_like*) – Source altitude in km
- **newheight** (*array_like*) – Destination altitude in km
- **conjugate** (*bool, optional*) – Map to *newheight* in the conjugate hemisphere instead of the closest hemisphere
- **precision** (*float, optional*) – Precision of output (degrees). A negative value of this argument produces a low-precision calculation of geodetic lat/lon based only on their spherical harmonic representation. A positive value causes the underlying Fortran routine to iterate until feeding the output geo lat/lon into geo2qd (APXG2Q) reproduces the input QD lat/lon to within the specified precision.

Returns

- **newglat** (*ndarray or float*) – Geodetic latitude of mapped point
- **newglon** (*ndarray or float*) – Geodetic longitude of mapped point
- **error** (*ndarray or float*) – The angular difference (degrees) between the input QD coordinates and the qlat/qlon produced by feeding the output glat and glon into geo2qd (APXG2Q)

Notes

The mapping is done by converting glat/glon/height to modified apex lat/lon, and converting back to geographic using newheight (if conjugate, use negative apex latitude when converting back)

`map_E_to_height`(*alat*, *alon*, *height*, *newheight*, *edata*)

Performs mapping of electric field along the magnetic field.

It is assumed that the electric field is perpendicular to B.

Parameters

- **alat** (*(N,) array_like or float*) – Modified apex latitude
- **alon** (*(N,) array_like or float*) – Modified apex longitude
- **height** (*(N,) array_like or float*) – Source altitude in km
- **newheight** (*(N,) array_like or float*) – Destination altitude in km

- **edata** (*(3,) or (3, N) array_like*) – Electric field (at *alat*, *alon*, *height*) in geodetic east, north, and up components

Returns

out (*(3, N) or (3,) ndarray*) – The electric field at *newheight* (geodetic east, north, and up components)

`map_V_to_height`(*alat*, *alon*, *height*, *newheight*, *vdata*)

Performs mapping of electric drift velocity along the magnetic field.

It is assumed that the electric field is perpendicular to B.

Parameters

- **alat** (*(N,) array_like or float*) – Modified apex latitude
- **alon** (*(N,) array_like or float*) – Modified apex longitude
- **height** (*(N,) array_like or float*) – Source altitude in km
- **newheight** (*(N,) array_like or float*) – Destination altitude in km
- **vdata** (*(3,) or (3, N) array_like*) – Electric drift velocity (at *alat*, *alon*, *height*) in geodetic east, north, and up components

Returns

out (*(3, N) or (3,) ndarray*) – The electric drift velocity at *newheight* (geodetic east, north, and up components)

`basevectors_qd`(*lat*, *lon*, *height*, *coords='geo'*, *precision=1e-10*)

Get quasi-dipole base vectors f1 and f2 at the specified coordinates.

Parameters

- **lat** (*(N,) array_like or float*) – Latitude
- **lon** (*(N,) array_like or float*) – Longitude
- **height** (*(N,) array_like or float*) – Altitude in km
- **coords** (*{'geo', 'apex', 'qd'}, optional*) – Input coordinate system
- **precision** (*float, optional*) – Precision of output (degrees) when converting to geo. A negative value of this argument produces a low-precision calculation of geodetic lat/lon based only on their spherical harmonic representation. A positive value causes the underlying Fortran routine to iterate until feeding the output geo lat/lon into geo2qd (APXG2Q) reproduces the input QD lat/lon to within the specified precision (all coordinates being converted to geo are converted to QD first and passed through APXG2Q).

Returns

- **f1** (*(2, N) or (2,) ndarray*)
- **f2** (*(2, N) or (2,) ndarray*)

Notes

The vectors are described by Richmond [1995][2] and Emmert et al. [2010][3]. The vector components are geodetic east and north.

References

basevectors_apex(*lat*, *lon*, *height*, *coords='geo'*, *precision=1e-10*)

Returns base vectors in quasi-dipole and apex coordinates.

Parameters

- **lat** (*array_like or float*) – Latitude in degrees; input must be broadcastable with *lon* and *height*.
- **lon** (*array_like or float*) – Longitude in degrees; input must be broadcastable with *lat* and *height*.
- **height** (*array_like or float*) – Altitude in km; input must be broadcastable with *lon* and *lat*.
- **coords** (*str, optional*) – Input coordinate system, expects one of 'geo', 'apex', or 'qd' (default='geo')
- **precision** (*float, optional*) – Precision of output (degrees) when converting to geo. A negative value of this argument produces a low-precision calculation of geodetic lat/lon based only on their spherical harmonic representation. A positive value causes the underlying Fortran routine to iterate until feeding the output geo lat/lon into geo2qd (APXG2Q) reproduces the input QD lat/lon to within the specified precision (all coordinates being converted to geo are converted to QD first and passed through APXG2Q).

Returns

- **f1** (*(3, N) or (3,) ndarray*) – Quasi-dipole base vector equivalent to e1, tangent to contours of constant lambdaA
- **f2** (*(3, N) or (3,) ndarray*) – Quasi-dipole base vector equivalent to e2
- **f3** (*(3, N) or (3,) ndarray*) – Quasi-dipole base vector equivalent to e3, tangent to contours of PhiA
- **g1** (*(3, N) or (3,) ndarray*) – Quasi-dipole base vector equivalent to d1
- **g2** (*(3, N) or (3,) ndarray*) – Quasi-dipole base vector equivalent to d2
- **g3** (*(3, N) or (3,) ndarray*) – Quasi-dipole base vector equivalent to d3
- **d1** (*(3, N) or (3,) ndarray*) – Apex base vector normal to contours of constant PhiA
- **d2** (*(3, N) or (3,) ndarray*) – Apex base vector that completes the right-handed system
- **d3** (*(3, N) or (3,) ndarray*) – Apex base vector normal to contours of constant lambdaA
- **e1** (*(3, N) or (3,) ndarray*) – Apex base vector tangent to contours of constant V0
- **e2** (*(3, N) or (3,) ndarray*) – Apex base vector that completes the right-handed system
- **e3** (*(3, N) or (3,) ndarray*) – Apex base vector tangent to contours of constant PhiA

[2] Richmond, A. D. (1995), Ionospheric Electrodynamics Using Magnetic Apex Coordinates, Journal of geomagnetism and geoelectricity, 47(2), 191–212, doi:10.5636/jgg.47.191.

[3] Emmert, J. T., A. D. Richmond, and D. P. Drob (2010), A computationally compact representation of Magnetic-Apex and Quasi-Dipole coordinates with smooth base vectors, J. Geophys. Res., 115(A8), A08322, doi:10.1029/2010JA015326.

Notes

The vectors are described by Richmond [1995][4] and Emmert et al. [2010][5]. The vector components are geodetic east, north, and up (only east and north for *f1* and *f2*).

f3, *g1*, *g2*, and *g3* are not part of the Fortran code by Emmert et al. [2010][5]. They are calculated by this Python library according to the following equations in Richmond [1995][4]:

- *g1*: Eqn. 6.3
- *g2*: Eqn. 6.4
- *g3*: Eqn. 6.5
- *f3*: Eqn. 6.8

References

`get_apex`(*lat*, *height=None*)

Calculate apex height

Parameters

- **lat** (*float*) – Apex latitude in degrees
- **height** (*float or NoneType*) – Height above the surface of the Earth in km or NoneType to use reference height (default=None)

Returns

apex_height (*float*) – Height of the field line apex in km

`get_height`(*lat*, *apex_height*)

Calculate the height given an apex latitude and apex height.

Parameters

- **lat** (*float*) – Apex latitude in degrees
- **apex_height** (*float*) – Maximum height of the apex field line above the surface of the Earth in km

Returns

height (*float*) – Height above the surface of the Earth in km

`set_epoch`(*year*)

Updates the epoch for all subsequent conversions.

Parameters

year (*float*) – Decimal year

`set_refh`(*refh*)

Updates the apex reference height for all subsequent conversions.

Parameters

refh (*float*) – Apex reference height in km

[4] Richmond, A. D. (1995), Ionospheric Electrodynamics Using Magnetic Apex Coordinates, Journal of geomagnetism and geoelectricity, 47(2), 191–212, doi:10.5636/jgg.47.191.

[5] Emmert, J. T., A. D. Richmond, and D. P. Drob (2010), A computationally compact representation of Magnetic-Apex and Quasi-Dipole coordinates with smooth base vectors, J. Geophys. Res., 115(A8), A08322, doi:10.1029/2010JA015326.

Notes

The reference height is the height to which field lines will be mapped, and is only relevant for conversions involving apex (not quasi-dipole).

`get_babs`(*glat*, *glon*, *height*)

Returns the magnitude of the IGRF magnetic field in tesla.

Parameters

- **glat** (*array_like*) – Geodetic latitude in degrees
- **glon** (*array_like*) – Geodetic longitude in degrees
- **height** (*array_like*) – Altitude in km

Returns

babs (*ndarray or float*) – Magnitude of the IGRF magnetic field in Tesla

`bvectors_apex`(*lat*, *lon*, *height*, *coords='geo'*, *precision=1e-10*)

Returns the magnetic field vectors in apex coordinates.

Parameters

- **lat** (*(N,) array_like or float*) – Latitude
- **lon** (*(N,) array_like or float*) – Longitude
- **height** (*(N,) array_like or float*) – Altitude in km
- **coords** (*{'geo', 'apex', 'qd'}, optional*) – Input coordinate system
- **precision** (*float, optional*) – Precision of output (degrees) when converting to geo. A negative value of this argument produces a low-precision calculation of geodetic lat/lon based only on their spherical harmonic representation. A positive value causes the underlying Fortran routine to iterate until feeding the output geo lat/lon into geo2qd (APXG2Q) reproduces the input QD lat/lon to within the specified precision (all coordinates being converted to geo are converted to QD first and passed through APXG2Q).

Returns

- **main_mag_e3** (*(1, N) or (1,) ndarray*) – IGRF magnitude divided by a scaling factor, D (d_scale) to give the main B field magnitude along the e3 base vector
- **e3** (*(3, N) or (3,) ndarray*) – Base vector tangent to the contours of constant V_0 and Phi_A
- **main_mag_d3** (*(1, N) or (1,) ndarray*) – IGRF magnitude multiplied by a scaling factor, D (d_scale) to give the main B field magnitudee along the d3 base vector
- **d3** (*(3, N) or (3,) ndarray*) – Base vector equivalent to the scaled main field unit vector

Notes

See Richmond, A. D. (1995)[4] equations 3.8-3.14

The apex magnetic field vectors described by Richmond [1995][Page 25, 4] and Emmert et al. [2010][Page 25, 5], specfically the Be3 (main_mag_e3) and Bd3 (main_mag_d3) components. The vector components are geodetic east, north, and up.

References

Richmond, A. D. (1995)[Page 25, 4] Emmert, J. T. et al. (2010)[Page 25, 5]

apexpy.helpers

This module contains helper functions used by *Apex*.

Module Contents

Functions

checklat(lat[, name])	Makes sure the latitude is inside [-90, 90], clipping close values
getsinIm(alat)	Computes sinIm from modified apex latitude.
getcosIm(alat)	Computes cosIm from modified apex latitude.
toYearFraction(date)	Converts `datetime.date` or `datetime.datetime` to decimal
gc2gdlat(gclat)	Converts geocentric latitude to geodetic latitude using WGS84.
subsol(datetime)	Finds subsolar geocentric latitude and longitude.

apexpy.helpers.**checklat**(*lat*, *name='lat'*)

Makes sure the latitude is inside [-90, 90], clipping close values (tolerance 1e-4).

Parameters

- **lat** (*array-like*) – latitude
- **name** (*str, optional*) – parameter name to use in the exception message

Returns

lat (*ndarray or float*) – Same as input where values just outside the range have been clipped to [-90, 90]

Raises

ValueError – if any values are too far outside the range [-90, 90]

apexpy.helpers.**getsinIm**(*alat*)

Computes sinIm from modified apex latitude.

Parameters

alat (*array-like*) – Modified apex latitude

Returns

sinIm (*ndarray or float*)

apexpy.helpers.**getcosIm**(*alat*)

Computes cosIm from modified apex latitude.

Parameters

alat (*array-like*) – Modified apex latitude

Returns

cosIm (*ndarray or float*)

apexpy.helpers.**toYearFraction**(*date*)

Converts `datetime.date` or `datetime.datetime` to decimal year.

Parameters
date (`datetime.date` or `datetime.datetime`) – Input date or datetime object

Returns
year (*float*) – Decimal year

Notes

The algorithm is taken from http://stackoverflow.com/a/6451892/2978652

apexpy.helpers.**gc2gdlat**(*gclat*)

Converts geocentric latitude to geodetic latitude using WGS84.

Parameters
gclat (*array-like*) – Geocentric latitude

Returns
gdlat (*ndarray or float*) – Geodetic latitude

apexpy.helpers.**subsol**(*datetime*)

Finds subsolar geocentric latitude and longitude.

Parameters
datetime (`datetime.datetime` or `numpy.ndarray[datetime64]`) – Date and time in UTC (naive objects are treated as UTC)

Returns

- **sbsllat** (*float*) – Latitude of subsolar point
- **sbsllon** (*float*) – Longitude of subsolar point

Notes

Based on formulas in Astronomical Almanac for the year 1996, p. C24. (U.S. Government Printing Office, 1994). Usable for years 1601-2100, inclusive. According to the Almanac, results are good to at least 0.01 degree latitude and 0.025 degrees longitude between years 1950 and 2050. Accuracy for other years has not been tested. Every day is assumed to have exactly 86400 seconds; thus leap seconds that sometimes occur on December 31 are ignored (their effect is below the accuracy threshold of the algorithm).

After Fortran code by A. D. Richmond, NCAR. Translated from IDL by K. Laundal.

Package Contents

Classes

Apex	Calculates coordinate conversions, field-line mapping, and base vectors.

class apexpy.**Apex**(*date=None*, *refh=0*, *datafile=None*, *fortranlib=None*)

Bases: object

Calculates coordinate conversions, field-line mapping, and base vectors.

Parameters

- **date** (NoneType, float, dt.date, or dt.datetime, optional) – Determines which IGRF coefficients are used in conversions. Uses current date as default. If float, use decimal year. If None, uses current UTC. (default=None)
- **refh** (*float, optional*) – Reference height in km for apex coordinates, the field lines are mapped to this height. (default=0)
- **datafile** (*str or NoneType, optional*) – Path to custom coefficient file, if None uses *apexsh.dat* file (default=None)
- **fortranlib** (*str or NoneType, optional*) – Path to Fortran Apex CPython library, if None uses linked library file (default=None)

Variables

- **year** (*float*) – Decimal year used for the IGRF model
- **RE** (*float*) – Earth radius in km, defaults to mean Earth radius
- **refh** (*float*) – Reference height in km for apex coordinates
- **datafile** (*str*) – Path to coefficient file
- **fortranlib** (*str*) – Path to Fortran Apex CPython library
- **igrf_fn** (*str*) – IGRF coefficient filename

Notes

The calculations use IGRF-13 with coefficients from 1900 to 2025[1].

The geodetic reference ellipsoid is WGS84.

References

__repr__()

Produce an evaluatable representation of the Apex class.

__str__()

Produce a user-friendly representation of the Apex class.

__eq__(*comp_obj*)

Performs equivalency evaluation.

Parameters
comp_obj – Object of any time to be compared to the class object

Returns
bool or NotImplemented – True if self and comp_obj are identical, False if they are not, and NotImplemented if the classes are not the same

[1] Thébault, E. et al. (2015), International Geomagnetic Reference Field: the 12th generation, Earth, Planets and Space, 67(1), 79, doi:10.1186/s40623-015-0228-9.

`_apex2qd_nonvectorized`(*alat*, *alon*, *height*)

Convert from apex to quasi-dipole (not-vectorised)

Parameters

- **alat** ((*float*)) – Apex latitude in degrees
- **alon** ((*float*)) – Apex longitude in degrees
- **height** ((*float*)) – Height in km

Returns

- **qlat** ((*float*)) – Quasi-dipole latitude in degrees
- **qlon** ((*float*)) – Quasi-diplole longitude in degrees

`_qd2apex_nonvectorized`(*qlat*, *qlon*, *height*)

Converts quasi-dipole to modified apex coordinates.

Parameters

- **qlat** (*float*) – Quasi-dipole latitude
- **qlon** (*float*) – Quasi-dipole longitude
- **height** (*float*) – Altitude in km

Returns

- **alat** (*float*) – Modified apex latitude
- **alon** (*float*) – Modified apex longitude

Raises

ApexHeightError – if apex height < reference height

`_map_EV_to_height`(*alat*, *alon*, *height*, *newheight*, *data*, *ev_flag*)

Maps electric field related values to a desired height

Parameters

- **alat** (*array-like*) – Apex latitude in degrees.
- **alon** (*array-like*) – Apex longitude in degrees.
- **height** (*array-like*) – Current altitude in km.
- **new_height** (*array-like*) – Desired altitude to which EV values will be mapped in km.
- **data** (*array-like*) – 3D value(s) for the electric field or electric drift
- **ev_flag** (*str*) – Specify if value is an electric field ('E') or electric drift ('V')

Returns

data_mapped (*array-like*) – Data mapped along the magnetic field from the old height to the new height.

Raises

`ValueError` – If an unknown *ev_flag* or badly shaped data input is supplied.

`_get_babs_nonvectorized`(*glat*, *glon*, *height*)

Get the absolute value of the B-field in Tesla

Parameters

- **glat** (*float*) – Geodetic latitude in degrees

- **glon** (*float*) – Geodetic longitude in degrees
- **height** (*float*) – Altitude in km

Returns

babs (*float*) – Absolute value of the magnetic field in Tesla

`convert`(*lat*, *lon*, *source*, *dest*, *height=0*, *datetime=None*, *precision=1e-10*, *ssheight=50 * 6371*)

Converts between geodetic, modified apex, quasi-dipole and MLT.

Parameters

- **lat** (*array_like*) – Latitude in degrees
- **lon** (*array_like*) – Longitude in degrees or MLT in hours
- **source** (*str*) – Input coordinate system, accepts 'geo', 'apex', 'qd', or 'mlt'
- **dest** (*str*) – Output coordinate system, accepts 'geo', 'apex', 'qd', or 'mlt'
- **height** (*array_like, optional*) – Altitude in km
- **datetime** (`datetime.datetime`) – Date and time for MLT conversions (required for MLT conversions)
- **precision** (*float, optional*) – Precision of output (degrees) when converting to geo. A negative value of this argument produces a low-precision calculation of geodetic lat/lon based only on their spherical harmonic representation. A positive value causes the underlying Fortran routine to iterate until feeding the output geo lat/lon into geo2qd (APXG2Q) reproduces the input QD lat/lon to within the specified precision (all coordinates being converted to geo are converted to QD first and passed through APXG2Q).
- **ssheight** (*float, optional*) – Altitude in km to use for converting the subsolar point from geographic to magnetic coordinates. A high altitude is used to ensure the subsolar point is mapped to high latitudes, which prevents the South-Atlantic Anomaly (SAA) from influencing the MLT.

Returns

- **lat** (*ndarray or float*) – Converted latitude (if converting to MLT, output latitude is apex)
- **lon** (*ndarray or float*) – Converted longitude or MLT

Raises

`ValueError` – For unknown source or destination coordinate system or a missing or badly formed latitude or datetime input

`geo2apex`(*glat*, *glon*, *height*)

Converts geodetic to modified apex coordinates.

Parameters

- **glat** (*array_like*) – Geodetic latitude
- **glon** (*array_like*) – Geodetic longitude
- **height** (*array_like*) – Altitude in km

Returns

- **alat** (*ndarray or float*) – Modified apex latitude
- **alon** (*ndarray or float*) – Modified apex longitude

`apex2geo`(*alat*, *alon*, *height*, *precision=1e-10*)

Converts modified apex to geodetic coordinates.

Parameters

- **alat** (*array_like*) – Modified apex latitude
- **alon** (*array_like*) – Modified apex longitude
- **height** (*array_like*) – Altitude in km
- **precision** (*float, optional*) – Precision of output (degrees). A negative value of this argument produces a low-precision calculation of geodetic lat/lon based only on their spherical harmonic representation. A positive value causes the underlying Fortran routine to iterate until feeding the output geo lat/lon into geo2qd (APXG2Q) reproduces the input QD lat/lon to within the specified precision.

Returns

- **glat** (*ndarray or float*) – Geodetic latitude
- **glon** (*ndarray or float*) – Geodetic longitude
- **error** (*ndarray or float*) – The angular difference (degrees) between the input QD coordinates and the qlat/qlon produced by feeding the output glat and glon into geo2qd (APXG2Q)

`geo2qd`(*glat*, *glon*, *height*)

Converts geodetic to quasi-dipole coordinates.

Parameters

- **glat** (*array_like*) – Geodetic latitude
- **glon** (*array_like*) – Geodetic longitude
- **height** (*array_like*) – Altitude in km

Returns

- **qlat** (*ndarray or float*) – Quasi-dipole latitude
- **qlon** (*ndarray or float*) – Quasi-dipole longitude

`qd2geo`(*qlat*, *qlon*, *height*, *precision=1e-10*)

Converts quasi-dipole to geodetic coordinates.

Parameters

- **qlat** (*array_like*) – Quasi-dipole latitude
- **qlon** (*array_like*) – Quasi-dipole longitude
- **height** (*array_like*) – Altitude in km
- **precision** (*float, optional*) – Precision of output (degrees). A negative value of this argument produces a low-precision calculation of geodetic lat/lon based only on their spherical harmonic representation. A positive value causes the underlying Fortran routine to iterate until feeding the output geo lat/lon into geo2qd (APXG2Q) reproduces the input QD lat/lon to within the specified precision.

Returns

- **glat** (*ndarray or float*) – Geodetic latitude
- **glon** (*ndarray or float*) – Geodetic longitude

- **error** (*ndarray or float*) – The angular difference (degrees) between the input QD coordinates and the qlat/qlon produced by feeding the output glat and glon into geo2qd (APXG2Q)

apex2qd(*alat*, *alon*, *height*)

Converts modified apex to quasi-dipole coordinates.

Parameters

- **alat** (*array_like*) – Modified apex latitude
- **alon** (*array_like*) – Modified apex longitude
- **height** (*array_like*) – Altitude in km

Returns

- **qlat** (*ndarray or float*) – Quasi-dipole latitude
- **qlon** (*ndarray or float*) – Quasi-dipole longitude

Raises

ApexHeightError – if *height* > apex height

qd2apex(*qlat*, *qlon*, *height*)

Converts quasi-dipole to modified apex coordinates.

Parameters

- **qlat** (*array_like*) – Quasi-dipole latitude
- **qlon** (*array_like*) – Quasi-dipole longitude
- **height** (*array_like*) – Altitude in km

Returns

- **alat** (*ndarray or float*) – Modified apex latitude
- **alon** (*ndarray or float*) – Modified apex longitude

Raises

ApexHeightError – if apex height < reference height

mlon2mlt(*mlon*, *dtime*, *ssheight=318550*)

Computes the magnetic local time at the specified magnetic longitude and UT.

Parameters

- **mlon** (*array_like*) – Magnetic longitude (apex and quasi-dipole longitude are always equal)
- **dtime** (`datetime.datetime`) – Date and time
- **ssheight** (*float, optional*) – Altitude in km to use for converting the subsolar point from geographic to magnetic coordinates. A high altitude is used to ensure the subsolar point is mapped to high latitudes, which prevents the South-Atlantic Anomaly (SAA) from influencing the MLT. The current default is 50 * 6371, roughly 50 RE. (default=318550)

Returns

mlt (*ndarray or float*) – Magnetic local time in hours [0, 24)

Notes

To compute the MLT, we find the apex longitude of the subsolar point at the given time. Then the MLT of the given point will be computed from the separation in magnetic longitude from this point (1 hour = 15 degrees).

`mlt2mlon`(*mlt*, *dtime*, *ssheight=318550*)

Computes the magnetic longitude at the specified MLT and UT.

Parameters

- **mlt** (*array_like*) – Magnetic local time
- **dtime** (`datetime.datetime`) – Date and time
- **ssheight** (*float, optional*) – Altitude in km to use for converting the subsolar point from geographic to magnetic coordinates. A high altitude is used to ensure the subsolar point is mapped to high latitudes, which prevents the South-Atlantic Anomaly (SAA) from influencing the MLT. The current default is 50 * 6371, roughly 50 RE. (default=318550)

Returns

mlon (*ndarray or float*) – Magnetic longitude [0, 360) (apex and quasi-dipole longitude are always equal)

Notes

To compute the magnetic longitude, we find the apex longitude of the subsolar point at the given time. Then the magnetic longitude of the given point will be computed from the separation in magnetic local time from this point (1 hour = 15 degrees).

`map_to_height`(*glat*, *glon*, *height*, *newheight*, *conjugate=False*, *precision=1e-10*)

Performs mapping of points along the magnetic field to the closest or conjugate hemisphere.

Parameters

- **glat** (*array_like*) – Geodetic latitude
- **glon** (*array_like*) – Geodetic longitude
- **height** (*array_like*) – Source altitude in km
- **newheight** (*array_like*) – Destination altitude in km
- **conjugate** (*bool, optional*) – Map to *newheight* in the conjugate hemisphere instead of the closest hemisphere
- **precision** (*float, optional*) – Precision of output (degrees). A negative value of this argument produces a low-precision calculation of geodetic lat/lon based only on their spherical harmonic representation. A positive value causes the underlying Fortran routine to iterate until feeding the output geo lat/lon into geo2qd (APXG2Q) reproduces the input QD lat/lon to within the specified precision.

Returns

- **newglat** (*ndarray or float*) – Geodetic latitude of mapped point
- **newglon** (*ndarray or float*) – Geodetic longitude of mapped point
- **error** (*ndarray or float*) – The angular difference (degrees) between the input QD coordinates and the qlat/qlon produced by feeding the output glat and glon into geo2qd (APXG2Q)

Notes

The mapping is done by converting glat/glon/height to modified apex lat/lon, and converting back to geographic using newheight (if conjugate, use negative apex latitude when converting back)

`map_E_to_height`(*alat*, *alon*, *height*, *newheight*, *edata*)

Performs mapping of electric field along the magnetic field.

It is assumed that the electric field is perpendicular to B.

Parameters

- **alat** (*(N,) array_like or float*) – Modified apex latitude
- **alon** (*(N,) array_like or float*) – Modified apex longitude
- **height** (*(N,) array_like or float*) – Source altitude in km
- **newheight** (*(N,) array_like or float*) – Destination altitude in km
- **edata** (*(3,) or (3, N) array_like*) – Electric field (at *alat*, *alon*, *height*) in geodetic east, north, and up components

Returns

out (*(3, N) or (3,) ndarray*) – The electric field at *newheight* (geodetic east, north, and up components)

`map_V_to_height`(*alat*, *alon*, *height*, *newheight*, *vdata*)

Performs mapping of electric drift velocity along the magnetic field.

It is assumed that the electric field is perpendicular to B.

Parameters

- **alat** (*(N,) array_like or float*) – Modified apex latitude
- **alon** (*(N,) array_like or float*) – Modified apex longitude
- **height** (*(N,) array_like or float*) – Source altitude in km
- **newheight** (*(N,) array_like or float*) – Destination altitude in km
- **vdata** (*(3,) or (3, N) array_like*) – Electric drift velocity (at *alat*, *alon*, *height*) in geodetic east, north, and up components

Returns

out (*(3, N) or (3,) ndarray*) – The electric drift velocity at *newheight* (geodetic east, north, and up components)

`basevectors_qd`(*lat*, *lon*, *height*, *coords='geo'*, *precision=1e-10*)

Get quasi-dipole base vectors f1 and f2 at the specified coordinates.

Parameters

- **lat** (*(N,) array_like or float*) – Latitude
- **lon** (*(N,) array_like or float*) – Longitude
- **height** (*(N,) array_like or float*) – Altitude in km
- **coords** (*{'geo', 'apex', 'qd'}, optional*) – Input coordinate system
- **precision** (*float, optional*) – Precision of output (degrees) when converting to geo. A negative value of this argument produces a low-precision calculation of geodetic lat/lon based only on their spherical harmonic representation. A positive value causes the underlying

Fortran routine to iterate until feeding the output geo lat/lon into geo2qd (APXG2Q) reproduces the input QD lat/lon to within the specified precision (all coordinates being converted to geo are converted to QD first and passed through APXG2Q).

Returns

- **f1** (*(2, N) or (2,) ndarray*)
- **f2** (*(2, N) or (2,) ndarray*)

Notes

The vectors are described by Richmond [1995][2] and Emmert et al. [2010][3]. The vector components are geodetic east and north.

References

basevectors_apex(*lat*, *lon*, *height*, *coords='geo'*, *precision=1e-10*)

Returns base vectors in quasi-dipole and apex coordinates.

Parameters

- **lat** (*array_like or float*) – Latitude in degrees; input must be broadcastable with *lon* and *height*.
- **lon** (*array_like or float*) – Longitude in degrees; input must be broadcastable with *lat* and *height*.
- **height** (*array_like or float*) – Altitude in km; input must be broadcastable with *lon* and *lat*.
- **coords** (*str, optional*) – Input coordinate system, expects one of 'geo', 'apex', or 'qd' (default='geo')
- **precision** (*float, optional*) – Precision of output (degrees) when converting to geo. A negative value of this argument produces a low-precision calculation of geodetic lat/lon based only on their spherical harmonic representation. A positive value causes the underlying Fortran routine to iterate until feeding the output geo lat/lon into geo2qd (APXG2Q) reproduces the input QD lat/lon to within the specified precision (all coordinates being converted to geo are converted to QD first and passed through APXG2Q).

Returns

- **f1** (*(3, N) or (3,) ndarray*) – Quasi-dipole base vector equivalent to e1, tangent to contours of constant lambdaA
- **f2** (*(3, N) or (3,) ndarray*) – Quasi-dipole base vector equivalent to e2
- **f3** (*(3, N) or (3,) ndarray*) – Quasi-dipole base vector equivalent to e3, tangent to contours of PhiA
- **g1** (*(3, N) or (3,) ndarray*) – Quasi-dipole base vector equivalent to d1
- **g2** (*(3, N) or (3,) ndarray*) – Quasi-dipole base vector equivalent to d2
- **g3** (*(3, N) or (3,) ndarray*) – Quasi-dipole base vector equivalent to d3
- **d1** (*(3, N) or (3,) ndarray*) – Apex base vector normal to contours of constant PhiA

[2] Richmond, A. D. (1995), Ionospheric Electrodynamics Using Magnetic Apex Coordinates, Journal of geomagnetism and geoelectricity, 47(2), 191–212, doi:10.5636/jgg.47.191.

[3] Emmert, J. T., A. D. Richmond, and D. P. Drob (2010), A computationally compact representation of Magnetic-Apex and Quasi-Dipole coordinates with smooth base vectors, J. Geophys. Res., 115(A8), A08322, doi:10.1029/2010JA015326.

- **d2** (*(3, N) or (3,) ndarray*) – Apex base vector that completes the right-handed system
- **d3** (*(3, N) or (3,) ndarray*) – Apex base vector normal to contours of constant lambdaA
- **e1** (*(3, N) or (3,) ndarray*) – Apex base vector tangent to contours of constant V0
- **e2** (*(3, N) or (3,) ndarray*) – Apex base vector that completes the right-handed system
- **e3** (*(3, N) or (3,) ndarray*) – Apex base vector tangent to contours of constant PhiA

Notes

The vectors are described by Richmond [1995][4] and Emmert et al. [2010][5]. The vector components are geodetic east, north, and up (only east and north for *f1* and *f2*).

f3, *g1*, *g2*, and *g3* are not part of the Fortran code by Emmert et al. [2010][Page 37, 5]. They are calculated by this Python library according to the following equations in Richmond [1995][4]:

- *g1*: Eqn. 6.3
- *g2*: Eqn. 6.4
- *g3*: Eqn. 6.5
- *f3*: Eqn. 6.8

References

`get_apex`(*lat*, *height=None*)

Calculate apex height

Parameters

- **lat** (*float*) – Apex latitude in degrees
- **height** (*float or NoneType*) – Height above the surface of the Earth in km or NoneType to use reference height (default=None)

Returns

apex_height (*float*) – Height of the field line apex in km

`get_height`(*lat*, *apex_height*)

Calculate the height given an apex latitude and apex height.

Parameters

- **lat** (*float*) – Apex latitude in degrees
- **apex_height** (*float*) – Maximum height of the apex field line above the surface of the Earth in km

Returns

height (*float*) – Height above the surface of the Earth in km

[4] Richmond, A. D. (1995), Ionospheric Electrodynamics Using Magnetic Apex Coordinates, Journal of geomagnetism and geoelectricity, 47(2), 191–212, doi:10.5636/jgg.47.191.

[5] Emmert, J. T., A. D. Richmond, and D. P. Drob (2010), A computationally compact representation of Magnetic-Apex and Quasi-Dipole coordinates with smooth base vectors, J. Geophys. Res., 115(A8), A08322, doi:10.1029/2010JA015326.

`set_epoch`(*year*)

Updates the epoch for all subsequent conversions.

Parameters
year (*float*) – Decimal year

`set_refh`(*refh*)

Updates the apex reference height for all subsequent conversions.

Parameters
refh (*float*) – Apex reference height in km

Notes

The reference height is the height to which field lines will be mapped, and is only relevant for conversions involving apex (not quasi-dipole).

`get_babs`(*glat*, *glon*, *height*)

Returns the magnitude of the IGRF magnetic field in tesla.

Parameters

- **glat** (*array_like*) – Geodetic latitude in degrees
- **glon** (*array_like*) – Geodetic longitude in degrees
- **height** (*array_like*) – Altitude in km

Returns
babs (*ndarray or float*) – Magnitude of the IGRF magnetic field in Tesla

`bvectors_apex`(*lat*, *lon*, *height*, *coords='geo'*, *precision=1e-10*)

Returns the magnetic field vectors in apex coordinates.

Parameters

- **lat** (*(N,) array_like or float*) – Latitude
- **lon** (*(N,) array_like or float*) – Longitude
- **height** (*(N,) array_like or float*) – Altitude in km
- **coords** (*{'geo', 'apex', 'qd'}, optional*) – Input coordinate system
- **precision** (*float, optional*) – Precision of output (degrees) when converting to geo. A negative value of this argument produces a low-precision calculation of geodetic lat/lon based only on their spherical harmonic representation. A positive value causes the underlying Fortran routine to iterate until feeding the output geo lat/lon into geo2qd (APXG2Q) reproduces the input QD lat/lon to within the specified precision (all coordinates being converted to geo are converted to QD first and passed through APXG2Q).

Returns

- **main_mag_e3** (*(1, N) or (1,) ndarray*) – IGRF magnitude divided by a scaling factor, D (d_scale) to give the main B field magnitude along the e3 base vector
- **e3** (*(3, N) or (3,) ndarray*) – Base vector tangent to the contours of constant V_0 and Phi_A
- **main_mag_d3** (*(1, N) or (1,) ndarray*) – IGRF magnitude multiplied by a scaling factor, D (d_scale) to give the main B field magnitudee along the d3 base vector
- **d3** (*(3, N) or (3,) ndarray*) – Base vector equivalent to the scaled main field unit vector

Notes

See Richmond, A. D. (1995)[4] equations 3.8-3.14

The apex magnetic field vectors described by Richmond [1995][Page 37, 4] and Emmert et al. [2010][Page 37, 5], specfically the Be3 (main_mag_e3) and Bd3 (main_mag_d3) components. The vector components are geodetic east, north, and up.

References

Richmond, A. D. (1995)[Page 37, 4] Emmert, J. T. et al. (2010)[Page 37, 5]

exception apexpy.**ApexHeightError**

Bases: `ValueError`

Specialized ValueError definition, to be used when apex height is wrong.

4.2 Command-line interface

When you install this package you will get a command called `apexpy`, which is an interface to the *convert()* method. See the documentation for this method for a more thorough explanation of arguments and behaviour.

You can get help on the command by running `apexpy -h`.

```
$ apexpy -h
usage: apexpy [-h] [--height HEIGHT] [--refh REFH] [-i FILE_IN]
              [-o FILE_OUT] SOURCE DEST DATE

Converts between geodetic, modified apex, quasi-dipole and MLT

positional arguments:
  SOURCE                Convert from {geo, apex, qd, mlt}
  DEST                  Convert to {geo, apex, qd, mlt}
  DATE                  YYYY[MM[DD[HHMMSS]]] date/time for IGRF
                        coefficients, time part required for MLT
                        calculations

optional arguments:
  -h, --help            show this help message and exit
  --height HEIGHT       height for conversion
  --refh REFH           reference height for modified apex coordinates
  -i FILE_IN, --input FILE_IN
                        input file (stdin if none specified)
  -o FILE_OUT, --output FILE_OUT
                        output file (stdout if none specified)
```

CHAPTER

FIVE

CONTRIBUTING

Bug reports, feature suggestions and other contributions are greatly appreciated! While I can't promise to implement everything, I will always try to respond in a timely manner.

5.1 Short version

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

5.2 Bug reports

When reporting a bug please include:

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

5.3 Feature requests and feedback

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

If you are proposing a feature:

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

5.4 Development

To set up *apexpy* for local development:

1. Fork apexpy on GitHub.
2. Clone your fork locally:

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

3. Create a branch for local development based off of the `develop` branch:

```
git checkout -b name-of-your-bugfix-or-feature origin/develop
```

   Now you can make your changes locally. Add tests for bugs and new features in the relevant test file in the `tests` directory. The tests are run with `pytest` and can be written as normal functions (starting with `test_`) containing a standard `assert` statement for testing output.

4. When you're done making changes, run `pytest` locally if you can:

```
python -m pytest
```

5. Commit your changes and push your branch to GitHub:

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

   The project now uses the NumPy acronyms for development workflow in the commit messages.

6. Submit a pull request through the GitHub website. Pull requests should be made to the `develop` branch. The continuous integration (CI) testing servers will automatically test the whole codebase, including your changes, for multiple versions of Python on both Windows and Linux.

5.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code, just make a pull request.

For merging, you should:

1. Include passing tests for your changes
2. Update/add documentation if relevant
3. Add a note to `CHANGELOG.rst` about the changes
4. Add yourself to `AUTHORS.rst` and `.zenodo.json` with your ORCiD

5.4.2 Style Guidelines

In general, apexpy follows PEP8 and numpydoc guidelines. PyTest is used to run the unit and integration tests, flake8 checks for style, and sphinx-build performs documentation tests. However, there are certain additional style elements that have been settled on to ensure the project maintains a consistent coding style:

- Line breaks should occur before a binary operator (ignoring flake8 W503)
- Preferably break long lines on open parentheses instead of using backslashes
- Use no more than 80 characters per line
- Several dependent packages have common nicknames, including:
  - `import datetime as dt`
  - `import numpy as np`
- Provide tests with informative failure statements and descriptive, one-line docstrings.

apexpy is working on modernizing its code style to adhere to these guidelines.

CHAPTER

SIX

PACKAGE MAINTENANCE

6.1 Providing Wheels with Releases

The Continuous Integration (CI) now saves wheels created for each tested Python version and computer Operating System (OS) as artifacts. When preparing a new PyPi release, these wheels may be downloaded from the release candidate. We currently don't include them, because the wheels only work when the installation environment mirrors the CI environment.

6.2 Updating IGRF

The International Geomagnetic Reference Field is regularly updated to reflect the most recent changes to the Terrestrial magnetic field. apexpy currently uses IRGF-13 coefficients, which are provided in the `apexpy/apexpy/igrf13coeff.txt` file. To change or update the magnetic field coefficients used by apexpy, you need to update the python code, then rerun the fortran program that builds `apexpy/apexpy/apexsh.dat`. This is what makes apexpy performant. For more details, see Emmert et al. [2010][1].

Assuming your new coefficient file has the same format, the process is simple:

1. Clone the repository or your fork of the repository (see *Contributing*).
2. Update `apexpy/apexpy/apex.py` variable `igrf_fn` by setting it equal to the new IGRF coefficient filename (`igrf13coeff.txt`, for example).
3. In `apexpy/fortranapex/checkapexsh.f90`, update the variable `igrffilein` to the new IGRF coefficent filename. Relative paths are allowed.
4. Modify `checkapexsh.f90` by adding the next 5 year epoch to the `epochgrid` variable and updating the `nepochgrid` variable as necessary. For example, if the newest IGRF coefficients are good up to 2025 and `epochgrid` only has up to the year 2020, then add 2025 to `epochgrid` and then increment `nepochgrid` by 1.
5. Execute the `apextest` binary to generate the new `apexsh.dat` file.
6. Update the unit tests in the class `TestApexMethodExtrapolateIGRF` in `apexpy/apexpy/tests/test_Apex.py` so that they check the methods are working correctly with dates after the latest IGRF epoch (i.e., if the latest epoch is 2020, set the test to initialize with the year 2025). You will have to update the hard-coded confirmation values used by these tests.
7. Commit all changes and create a pull request on GitHub to integrate your branch with updated IGRF into the main repository.

[1] Emmert, J. T., A. D. Richmond, and D. P. Drob (2010), A computationally compact representation of Magnetic-Apex and Quasi-Dipole coordinates with smooth base vectors, J. Geophys. Res., 115(A8), A08322, doi:10.1029/2010JA015326.

6.3 Modifying Fortran Source

When modifying the fortran source code, it can be helpful to run a preliminary validation of the fortran output independent of the python wrapper. This should be done within the `apexpy/fortranapex` directory.

1. Remove any existing binaries by running the `make clean` command.
2. Build the `apextest` binary by running the `make` command.
3. Execute the `apextext` binary.
4. Confirm the output printed to the screen matches the test output shown in the comment block at the bottom of `checkapexsh.f90`. The output may not match the test output exactly due to floating point errors and improvements in the precision of the calculation.
5. If the modifications involved adding or removing fortran source files, modify the list of extension sources in `setup.cfg`.
6. Rebuild and install apexpy following the instructions in *Build from Source*.

6.4 Updating tests and style standards

apexpy is in the process of updating unit and integration tests to reduce code duplication and implementing cleaner style standards. Additionally, some parts of the fortran code adhere to older coding standards and raise warnings when compiled with newer compilers. If you would like to assist in these efforts (help would be appreciated), please discuss your potential contribution with the current maintainer to ensure a minimal duplication of effort.

CHAPTER

SEVEN

AUTHORS

This python wrapper is made by:

- Karl M. Laundal
- Christer van der Meeren
- Angeline G. Burrell (maintainer)
- Leslie Lamarche
- Gregory Starr
- Ashton Reimer
- Achim Morschhauser

Fortran code by Emmert et al. [2010][1]. Quasi-dipole and modified apex coordinates are defined by Richmond [1995][2]. The code uses IGRF-12 with coefficients valid through 2020 [Thébault et al., 2015][3].

[1] Emmert, J. T., A. D. Richmond, and D. P. Drob (2010), A computationally compact representation of Magnetic-Apex and Quasi-Dipole coordinates with smooth base vectors, J. Geophys. Res., 115(A8), A08322, doi:10.1029/2010JA015326.

[2] Richmond, A. D. (1995), Ionospheric Electrodynamics Using Magnetic Apex Coordinates, Journal of geomagnetism and geoelectricity, 47(2), 191–212, doi:10.5636/jgg.47.191.

[3] Thébault, E. et al. (2015), International Geomagnetic Reference Field: the 12th generation, Earth, Planets and Space, 67(1), 79, doi:10.1186/s40623-015-0228-9.

CHAPTER

EIGHT

CHANGELOG

8.1 2.0.1 (2023-04-11)

- Expanded installation instructions in the documenation
- Added unit tests for todays date, ensuring that *apex.dat* is current
- Added cron unit test to GitHub Actions CI
- Added a logo
- Correct indexing bug in Fortran source that was causing array overflow and memory errors for extrapolated years beyond the latest formal IGRF fit

8.2 2.0.0 (2022-12-09)

- Update Fortran source code to Fortran 90 standards
- Removed Python 2 support
- Updated community and package documentation
- Updated unit test style to reduce duplication and better follow PEP8
- Updated code style using codacy suggestions and reduced complexity
- Added class representation strings to Apex
- Improved input testing for Apex methods
- Added more examples and installation help to the documentation
- Fixed missing microseconds bug in helpers.subsol
- Added function to calculate height along a field line
- Changed installation to use meson
- Added wheel creation to CI
- Updated flake8 ignore syntax

8.3 1.1.0 (2021-03-05)

- Adapted Fortran to read IRGF coefficients from a file (updated to IGRF-13)
- Improved the subsol routine to allow array input
- Improved PEP8 compliance
- Added some missing docstrings to unit tests
- Fixed AppVeyor test environment
- Updated python test versions
- Updated community and package documentation
- Fixed bug where NaNs caused array input to crash
- Fixed bug in quasi-dipole to apex conversion at equator
- Removed duplicate CI services

8.4 1.0.4 (2019-04-05)

- Updated installation instructions
- Simplified warning tests
- Made some PEP8 changes

8.5 1.0.3 (2018-04-05)

- Updated badges and added DOI
- Added tests for python 3.6
- Removed tests for python 3.3
- Made some PEP8 changes

8.6 1.0.2 (2018-02-27)

- Extend character limit for allowable data file path, and update documentation to reflect a change in maintainers. Also updated testing implimentation, reduced fortran compiler warnings, and improved PEP8 compliance.

8.7 1.0.1 (2016-03-10)

- Remove geocentric to geodetic conversion of subsolar point based on feedback from Art Richmond. (The subsolar point is the same in geocentric and geodetic coordinates.) The helper function *gc2gdlat* have been kept to preserve backwards compatibility.

8.8 1.0.0 (2015-11-30)

- Initial release

CHAPTER

NINE

API REFERENCE

This page contains auto-generated API reference documentation[1].

9.1 test_helpers

Test the apexpy.helper submodule

Notes

Whenever function outputs are tested against hard-coded numbers, the test results (numbers) were obtained by running the code that is tested. Therefore, these tests below only check that nothing changes when refactoring, etc., and not if the results are actually correct.

These results are expected to change when IGRF is updated.

9.1.1 Module Contents

Classes

TestHelpers	Test class for the helper sub-module.

Functions

datetime64_to_datetime(dt64)	Convert numpy datetime64 object to a datetime datetime object.

test_helpers.**datetime64_to_datetime**(*dt64*)

Convert numpy datetime64 object to a datetime datetime object.

Parameters
dt64 (*np.datetime64*) – Numpy datetime64 object

Returns
dt.datetime – Equivalent datetime object with a resolution of days

[1] Created with sphinx-autoapi

Notes

Works outside 32 bit int second range of 1970

class test_helpers.**TestHelpers**

Bases: object

Test class for the helper sub-module.

setup_method()

Set up a clean test environment.

teardown_method()

Clean up the test environment.

eval_output(*rtol=1e-07*, *atol=0.0*)

Evaluate the values and shape of the calculated and expected output.

test_checklat_scalar(*lat*)

Test good latitude check with scalars.

Parameters
lat (*int or float*) – Latitude in degrees N

test_checklat_scalar_clip(*lat*)

Test good latitude check with scalars just beyond the lat limits.

Parameters
lat (*int or float*) – Latitude in degrees N

test_checklat_error(*in_args*, *msg*)

Test bad latitude raises ValueError with appropriate message.

Parameters

- **in_args** (*list*) – List of input arguments
- **msg** (*str*) – Expected error message

test_checklat_array(*lat*, *test_lat*)

Test good latitude with finite values.

Parameters

- **lat** (*array-like*) – Latitudes in degrees N
- **test_lat** (*list-like*) – Output latitudes in degrees N

test_getsinIm(*lat*, *test_sin*)

Test sin(Im) calculation for scalar and array inputs.

Parameters

- **lat** (*float*) – Latitude in degrees N
- **test_sin** (*float*) – Output value

test_getcosIm(*lat*, *test_cos*)

Test cos(Im) calculation for scalar and array inputs.

Parameters

- **lat** (*float*) – Latitude in degrees N

- **test_cos** (*float*) – Expected output

`test_toYearFraction`(*in_time*, *year*)

Test the datetime to fractional year calculation.

Parameters

- **in_time** (*dt.datetime or dt.date*) – Input time in a datetime format
- **year** (*int or float*) – Output year with fractional values

`test_gc2gdlat`(*gc_lat*, *gd_lat*)

Test geocentric to geodetic calculation.

Parameters

- **gc_lat** (*int or float*) – Geocentric latitude in degrees N
- **gd_lat** (*int or float*) – Geodetic latitude in degrees N

`test_subsol`(*in_time*, *test_loc*)

Test the subsolar location calculation.

Parameters

- **in_time** (*dt.datetime*) – Input time
- **test_loc** (*tuple*) – Expected output

`test_bad_subsol_date`(*in_time*)

Test raises ValueError for bad time in subsolar calculation.

Parameters
in_time (*dt.datetime*) – Input time

`test_bad_subsol_input`(*in_time*)

Test raises ValueError for bad input type in subsolar calculation.

Parameters
in_time (*NoneType or float*) – Badly formatted input time

`test_subsol_datetime64_array`(*in_dates*)

Verify subsolar point calculation using an array of np.datetime64.

Parameters
in_time (*array-like*) – Array of input times

Notes

Tested by ensuring the array of np.datetime64 is equivalent to converting using single dt.datetime values

9.2 test_Apex

Test the apexpy.Apex class

Notes

Whenever function outputs are tested against hard-coded numbers, the test results (numbers) were obtained by running the code that is tested. Therefore, these tests below only check that nothing changes when refactoring, etc., and not if the results are actually correct.

These results are expected to change when IGRF is updated.

9.2.1 Module Contents

Classes

TestApexInit	Test class for the Apex class object.
TestApexMethod	Test the Apex methods.
TestApexMLTMethods	Test the Apex Magnetic Local Time (MLT) methods.
TestApexMapMethods	Test the Apex height mapping methods.
TestApexBasevectorMethods	Test the Apex height base vector methods.
TestApexGetMethods	Test the Apex *get* methods.
TestApexMethodExtrapolateIGRF	Test the Apex methods on a year when IGRF must be extrapolated.

Functions

igrf_file([max_attempts])	A fixture for handling the coefficient file.
test_set_epoch_file_error(igrf_file)	Test raises OSError when IGRF coefficient file is missing.

test_Apex.**igrf_file**(*max_attempts=100*)

A fixture for handling the coefficient file.

Parameters
max_attempts (*int*) – Maximum rename attemps, needed for Windows (default=100)

test_Apex.**test_set_epoch_file_error**(*igrf_file*)

Test raises OSError when IGRF coefficient file is missing.

class test_Apex.**TestApexInit**

Bases: object

Test class for the Apex class object.

setup_method()

Initialize all tests.

teardown_method()

Clean up after each test.

eval_date()

Evaluate the times in self.test_date and self.apex_out.

eval_refh()

Evaluate the reference height in self.refh and self.apex_out.

test_init_defaults()

Test Apex class default initialization.

test_init_today()

Test Apex class initialization with today's date.

test_init_date(*in_date*)

Test Apex class with date initialization.

Parameters
in_date (*int, float, dt.date, or dt.datetime*) – Input date in a variety of formats

test_set_epoch(*new_date*)

Test successful setting of Apex epoch after initialization.

Parameters
new_date (*int or float*) – New date for the Apex class

test_init_refh(*in_refh*)

Test Apex class with reference height initialization.

Parameters
in_refh (*float*) – Input reference height in km

test_set_refh(*new_refh*)

Test the method used to set the reference height after the init.

Parameters
new_refh (*float*) – Reference height in km

test_init_with_bad_datafile()

Test raises IOError with non-existent datafile input.

test_init_with_bad_fortranlib()

Test raises IOError with non-existent datafile input.

test_repr_eval()

Test the Apex.__repr__ results.

test_ne_other_class()

Test Apex class inequality to a different class.

test_ne_missing_attr()

Test Apex class inequality when attributes are missing from one.

test_eq_missing_attr()

Test Apex class equality when attributes are missing from both.

test_str_eval()

Test the Apex.__str__ results.

class test_Apex.**TestApexMethod**

Bases: object

Test the Apex methods.

setup_method()

Initialize all tests.

teardown_method()

Clean up after each test.

get_input_args(*method_name*, *precision=0.0*)

Set the input arguments for the different Apex methods.

Parameters

- **method_name** (*str*) – Name of the Apex class method
- **precision** (*float*) – Value for the precision (default=0.0)

Returns

in_args (*list*) – List of the appropriate input arguments

test_apex_conversion_today()

Test Apex class conversion with today's date.

test_fortran_scalar_input(*apex_method*, *fortran_method*, *fslice*, *lat*, *lon*)

Tests Apex/fortran interface consistency for scalars.

Parameters

- **apex_method** (*str*) – Name of the Apex class method to test
- **fortran_method** (*str*) – Name of the Fortran function to test
- **fslice** (*slice*) – Slice used select the appropriate Fortran outputs
- **lat** (*int or float*) – Latitude in degrees N
- **lon** (*int or float*) – Longitude in degrees E

test_fortran_longitude_rollover(*apex_method*, *fortran_method*, *fslice*, *lat*, *lon1*, *lon2*)

Tests Apex/fortran interface consistency for longitude rollover.

Parameters

- **apex_method** (*str*) – Name of the Apex class method to test
- **fortran_method** (*str*) – Name of the Fortran function to test
- **fslice** (*slice*) – Slice used select the appropriate Fortran outputs
- **lat** (*int or float*) – Latitude in degrees N
- **lon1** (*int or float*) – Longitude in degrees E
- **lon2** (*int or float*) – Equivalent longitude in degrees E

test_fortran_array_input(*arr_shape*, *apex_method*, *fortran_method*, *fslice*)

Tests Apex/fortran interface consistency for array input.

Parameters

- **arr_shape** (*tuple*) – Expected output shape
- **apex_method** (*str*) – Name of the Apex class method to test

- **fortran_method** (*str*) – Name of the Fortran function to test
- **fslice** (*slice*) – Slice used select the appropriate Fortran outputs

`test_geo2apexall_scalar`(*lat*, *lon*)

Test Apex/fortran geo2apexall interface consistency for scalars.

Parameters

- **lat** (*int or float*) – Latitude in degrees N
- **long** (*int or float*) – Longitude in degrees E

`test_geo2apexall_array`(*arr_shape*)

Test Apex/fortran geo2apexall interface consistency for arrays.

Parameters

arr_shape (*tuple*) – Expected output shape

`test_convert_consistency`(*in_coord*, *out_coord*)

Test the self-consistency of the Apex convert method.

Parameters

- **in_coord** (*str*) – Input coordinate system
- **out_coord** (*str*) – Output coordinate system

`test_convert_at_lat_boundary`(*bound_lat*, *in_coord*, *out_coord*)

Test the conversion at the latitude boundary, with allowed excess.

Parameters

- **bound_lat** (*int or float*) – Boundary latitude in degrees N
- **in_coord** (*str*) – Input coordinate system
- **out_coord** (*str*) – Output coordinate system

`test_convert_qd2apex_at_equator`()

Test the quasi-dipole to apex conversion at the magnetic equator.

`test_convert_withnan`(*src*, *dest*)

Test Apex.convert success with NaN input.

Parameters

- **src** (*str*) – Input coordinate system
- **dest** (*str*) – Output coordinate system

`test_convert_invalid_lat`(*bad_lat*)

Test convert raises ValueError for invalid latitudes.

Parameters

bad_lat (*int or float*) – Latitude ouside the supported range in degrees N

`test_convert_invalid_transformation`(*coords*)

Test raises NotImplementedError for bad coordinates.

Parameters

coords (*tuple*) – Tuple specifying the input and output coordinate systems

`test_method_scalar_input`(*method_name*, *out_comp*)

Test the user method against set values with scalars.

Parameters

- **method_name** (*str*) – Apex class method to be tested
- **out_comp** (*tuple of floats*) – Expected output values

`test_method_broadcast_input`(*in_coord*, *out_coord*, *method_args*, *out_shape*)

Test the user method with inputs that require some broadcasting.

Parameters

- **in_coord** (*str*) – Input coordiante system
- **out_coord** (*str*) – Output coordiante system
- **method_args** (*list*) – List of input arguments
- **out_shape** (*tuple*) – Expected shape of output values

`test_method_invalid_lat`(*in_coord*, *out_coord*, *bad_lat*)

Test convert raises ValueError for invalid latitudes.

Parameters

- **in_coord** (*str*) – Input coordiante system
- **out_coord** (*str*) – Output coordiante system
- **bad_lat** (*int*) – Latitude in degrees N that is out of bounds

`test_method_at_lat_boundary`(*in_coord*, *out_coord*, *bound_lat*)

Test user methods at the latitude boundary, with allowed excess.

Parameters

- **in_coord** (*str*) – Input coordiante system
- **out_coord** (*str*) – Output coordiante system
- **bad_lat** (*int*) – Latitude in degrees N that is at the limits of the boundary

`test_geo2apex_undefined_warning`()

Test geo2apex warning and fill values for an undefined location.

`test_quasidipole_apexheight_close`(*method_name*, *delta_h*)

Test quasi-dipole success with a height close to the reference.

Parameters

- **method_name** (*str*) – Apex class method name to be tested
- **delta_h** (*float*) – tolerance for height in km

`test_quasidipole_raises_apexheight`(*method_name*, *hinc*, *msg*)

Quasi-dipole raises ApexHeightError when height above reference.

Parameters

- **method_name** (*str*) – Apex class method name to be tested
- **hinc** (*float*) – Height increment in km
- **msg** (*str*) – Expected output message

class test_Apex.**TestApexMLTMethods**

Bases: object

Test the Apex Magnetic Local Time (MLT) methods.

setup_method()

Initialize all tests.

teardown_method()

Clean up after each test.

test_convert_to_mlt(*in_coord*)

Test the conversions to MLT using Apex convert.

Parameters
in_coord (*str*) – Input coordinate system

test_convert_mlt_to_lon(*out_coord*)

Test the conversions from MLT using Apex convert.

Parameters
out_coord (*str*) – Output coordinate system

test_convert_geo2mlt_nodate()

Test convert from geo to MLT raises ValueError with no datetime.

test_mlon2mlt_scalar_inputs(*mlon_kwargs*, *test_mlt*)

Test mlon2mlt with scalar inputs.

Parameters

- **mlon_kwargs** (*dict*) – Input kwargs
- **test_mlt** (*float*) – Output MLT in hours

test_mlt2mlon_scalar_inputs(*mlt_kwargs*, *test_mlon*)

Test mlt2mlon with scalar inputs.

Parameters

- **mlt_kwargs** (*dict*) – Input kwargs
- **test_mlon** (*float*) – Output longitude in degrees E

test_mlon2mlt_array(*mlon*, *test_mlt*)

Test mlon2mlt with array inputs.

Parameters

- **mlon** (*array-like*) – Input longitudes in degrees E
- **test_mlt** (*float*) – Output MLT in hours

test_mlt2mlon_array(*mlt*, *test_mlon*)

Test mlt2mlon with array inputs.

Parameters

- **mlt** (*array-like*) – Input MLT in hours
- **test_mlon** (*float*) – Output longitude in degrees E

test_mlon2mlt_diffdates(*method_name*)

Test that MLT varies with universal time.

Parameters
method_name (*str*) – Name of Apex class method to be tested

test_mlon2mlt_offset(*mlt_offset*)

Test the time wrapping logic for the MLT.

Parameters
mlt_offset (*float*) – MLT offset in hours

test_mlt2mlon_offset(*mlon_offset*)

Test the time wrapping logic for the magnetic longitude.

Parameters
mlt_offset (*float*) – MLT offset in hours

test_convert_and_return(*order*, *start_val*)

Test the conversion to magnetic longitude or MLT and back again.

Parameters

- **order** (*list*) – List of strings specifying the order to run functions
- **start_val** (*int or float*) – Input value

class test_Apex.TestApexMapMethods

Bases: object

Test the Apex height mapping methods.

setup_method()

Initialize all tests.

teardown_method()

Clean up after each test.

test_map_to_height(*in_args*, *test_mapped*)

Test the map_to_height function.

Parameters

- **in_args** (*list*) – List of input arguments
- **test_mapped** (*list*) – List of expected outputs

test_map_to_height_same_height()

Test the map_to_height function when mapping to same height.

test_map_to_height_array_location(*arr_shape*, *ivec*)

Test map_to_height with array input.

Parameters

- **arr_shape** (*tuple*) – Expected array shape
- **ivec** (*int*) – Input argument index for vectorized input

test_mapping_height_raises_ApexHeightError(*method_name*, *in_args*)

Test map_to_height raises ApexHeightError.

Parameters

- **method_name** (*str*) – Name of the Apex class method to test
- **in_args** (*list*) – List of input arguments

`test_mapping_EV_bad_shape`(*method_name*, *ev_input*)

Test height mapping of E/V with baddly shaped input raises Error.

Parameters

- **method_name** (*str*) – Name of the Apex class method to test
- **ev_input** (*list*) – E/V input arguments

`test_mapping_EV_bad_flag`()

Test _map_EV_to_height raises error for bad data type flag.

`test_map_E_to_height_scalar_location`(*in_args*, *test_mapped*)

Test mapping of E-field to a specified height.

Parameters

- **in_args** (*list*) – List of input arguments
- **test_mapped** (*list*) – List of expected outputs

`test_map_EV_to_height_array_location`(*ev_flag*, *test_mapped*, *arr_shape*, *ivec*)

Test mapping of E-field/drift to a specified height with arrays.

Parameters

- **ev_flag** (*str*) – Character flag specifying whether to run 'E' or 'V' methods
- **test_mapped** (*list*) – List of expected outputs
- **arr_shape** (*tuple*) – Shape of the expected output
- **ivec** (*int*) – Index of the expected output

`test_map_V_to_height_scalar_location`(*in_args*, *test_mapped*)

Test mapping of velocity to a specified height.

Parameters

- **in_args** (*list*) – List of input arguments
- **test_mapped** (*list*) – List of expected outputs

`class test_Apex.TestApexBasevectorMethods`

Bases: `object`

Test the Apex height base vector methods.

`setup_method`()

Initialize all tests.

`teardown_method`()

Clean up after each test.

`get_comparison_results`(*bv_coord*, *coords*, *precision*)

Get the base vector results using the hidden function for comparison.

Parameters

- **bv_coord** (*str*) – Basevector coordinate scheme, expects on of 'apex', 'qd', or 'bvectors_apex'

- **coords** (*str*) – Expects one of 'geo', 'apex', or 'qd'
- **precision** (*float*) – Float specifiying precision

`test_basevectors_scalar`(*bv_coord*, *coords*, *precision*)

Test the base vector calculations with scalars.

Parameters

- **bv_coord** (*str*) – Name of the input coordinate system
- **coords** (*str*) – Name of the output coordinate system
- **precision** (*float*) – Level of run precision requested

`test_basevectors_scalar_shape`(*bv_coord*)

Test the shape of the scalar output.

Parameters

bv_coord (*str*) – Name of the input coordinate system

`test_basevectors_array`(*arr_shape*, *bv_coord*, *ivec*)

Test the output shape for array inputs.

Parameters

- **arr_shape** (*tuple*) – Expected output shape
- **bv_coord** (*str*) – Name of the input coordinate system
- **ivec** (*int*) – Index of the evaluated output value

`test_bvectors_apex`(*coords*)

Test the bvectors_apex method.

Parameters

coords (*str*) – Name of the coordiante system

`test_basevectors_apex_extra_values`()

Test specific values in the apex base vector output.

`test_basevectors_apex_delta`(*lat*, *lon*)

Test that vectors are calculated correctly.

Parameters

- **lat** (*int or float*) – Latitude in degrees N
- **lon** (*int or float*) – Longitude in degrees E

`test_basevectors_apex_invalid_scalar`()

Test warning and fill values for base vectors with bad inputs.

class test_Apex.**TestApexGetMethods**

Bases: `object`

Test the Apex *get* methods.

`setup_method`()

Initialize all tests.

`teardown_method`()

Clean up after each test.

`test_get_apex`(*alat*, *aheight*)

Test the apex height retrieval results.

Parameters

- **alat** (*int or float*) – Apex latitude in degrees N
- **aheight** (*int or float*) – Apex height in km

`test_get_babs`(*glat*, *glon*, *height*, *test_bmag*)

Test the method to get the magnitude of the magnetic field.

Parameters

- **glat** (*list*) – List of latitudes in degrees N
- **glon** (*list*) – List of longitudes in degrees E
- **height** (*list*) – List of heights in km
- **test_bmag** (*float*) – Expected B field magnitude

`test_get_apex_with_invalid_lat`(*bad_lat*)

Test get methods raise ValueError for invalid latitudes.

Parameters

bad_lat (*int or float*) – Bad input latitude in degrees N

`test_get_babs_with_invalid_lat`(*bad_lat*)

Test get methods raise ValueError for invalid latitudes.

Parameters

bad_lat (*int or float*) – Bad input latitude in degrees N

`test_get_at_lat_boundary`(*bound_lat*)

Test get methods at the latitude boundary, with allowed excess.

Parameters

bound_lat (*int or float*) – Boundary input latitude in degrees N

`test_get_height_at_equator`(*apex_height*)

Test that *get_height* returns apex height at equator.

Parameters

apex_height (*float*) – Apex height

`test_get_height_along_fieldline`(*lat*, *height*)

Test that *get_height* returns expected height of field line.

Parameters

- **lat** (*float*) – Input latitude
- **height** (*float*) – Output field-line height for line with apex of 3000 km

class test_Apex.**TestApexMethodExtrapolateIGRF**

Bases: `object`

Test the Apex methods on a year when IGRF must be extrapolated.

Notes

Extrapolation should be done using a year within 5 years of the latest IGRF model epoch.

setup_method()

Initialize all tests.

teardown_method()

Clean up after each test.

test_method_scalar_input(*method_name*, *out_comp*)

Test the user method against set values with scalars.

Parameters

- **method_name** (*str*) – Apex class method to be tested
- **out_comp** (*tuple of floats*) – Expected output values

test_convert_to_mlt()

Test conversion from mlon to mlt with scalars.

9.3 test_cmd

Unit tests for command line execution.

9.3.1 Module Contents

Classes

TestCommandLine	Test class for the command-line apexpy interface.

class test_cmd.**TestCommandLine**

Bases: object

Test class for the command-line apexpy interface.

setup_method()

Runs before every test method to create a clean environment.

teardown_method()

Runs after every method to clean up previous testing.

execute_command_line(*command*, *command_kwargs=None*, *pipe_out=False*)

Execute the command and load data from self.outfile

Parameters

- **command** (*list or str*) – List or string containing command to execute using subprocess
- **command_kwargs** (*dict or NoneType*) – Dict containing optional kwargs for subprocess.Popen command or None if using defaults. (default=None)
- **pipe_out** (*bool*) – Return pipe output instead of output from a data file if True (default=False)

Returns
data (*np.array, NoneType, or subprocess.Popen attribute*) – Numpy array of data from output file, None if no file was created, or the requested output from the pipe command.

`test_convert_w_datetime`(*date_str*)

Test command line with different date and time specification.

Parameters
date_str (*str*) – Input date string

`test_convert_single_line`()

Test command line with a single line of output.

`test_convert_stdin_stdout_w_height_flags`(*height*, *out_list*)

Test use of pipe input to command-line call with height flags.

Parameters

- **height** (*str*) – String specifying height with command line options
- **out_list** (*list*) – List of expected output values

`test_convert_mlt`()

Test magnetic local time conversion.

`test_invalid_date`(*date_str*)

Test raises ValueError with an invalid input date.

Parameters
date_str (*str*) – Input date string

`test_mlt_nodatetime`()

Test raises ValueError when time not provided for MLT calc.

`test_invalid_coord`(*coords*)

Test raises error when bad coordinate input provided.

Parameters
coords (*str*) – Input/output coordinate pairs

9.4 test_fortranapex

Test the apexpy.fortranapex class

Notes

Whenever function outputs are tested against hard-coded numbers, the test results (numbers) were obtained by running the code that is tested. Therefore, these tests below only check that nothing changes when refactoring, etc., and not if the results are actually correct.

These results are expected to change when IGRF is updated.

9.4.1 Module Contents

Classes

TestFortranApex	Test class for the Python-wrapped Fortran functions.

class test_fortranapex.**TestFortranApex**

Bases: object

Test class for the Python-wrapped Fortran functions.

setup_method()

Initialize each test.

teardown_method()

Clean environment after each test.

run_test_evaluation(*rtol=1e-05*, *atol=1e-05*)

Run the evaluation of the test results.

Parameters

- **rtol** (*float*) – Relative tolerance, default value based on old code's precision (default=1e-5)
- **atol** (*float*) – Absolute tolerance, default value based on old code's precision (default=1e-5)

test_apxg2q()

Test fortran apex geographic to quasi-dipole.

test_apxg2all()

Test fortran apex geographic to all outputs.

test_apxq2g()

Test fortran quasi-dipole to geographic.

test_g2q2d(*lat*, *lon*)

Test fortran geographic to quasi-dipole and back again.

Parameters

- **lat** (*int or float*) – Latitude in degrees N
- **lon** (*int or float*) – Longitude in degrees E

test_apxq2g_lowprecision()

Test low precision error value.

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

t

INDEX

Symbols

A

B

C

D

E

G