PlanetMapper

Release 1.9.1

Oliver King

Apr 24, 2024

GETTING STARTED

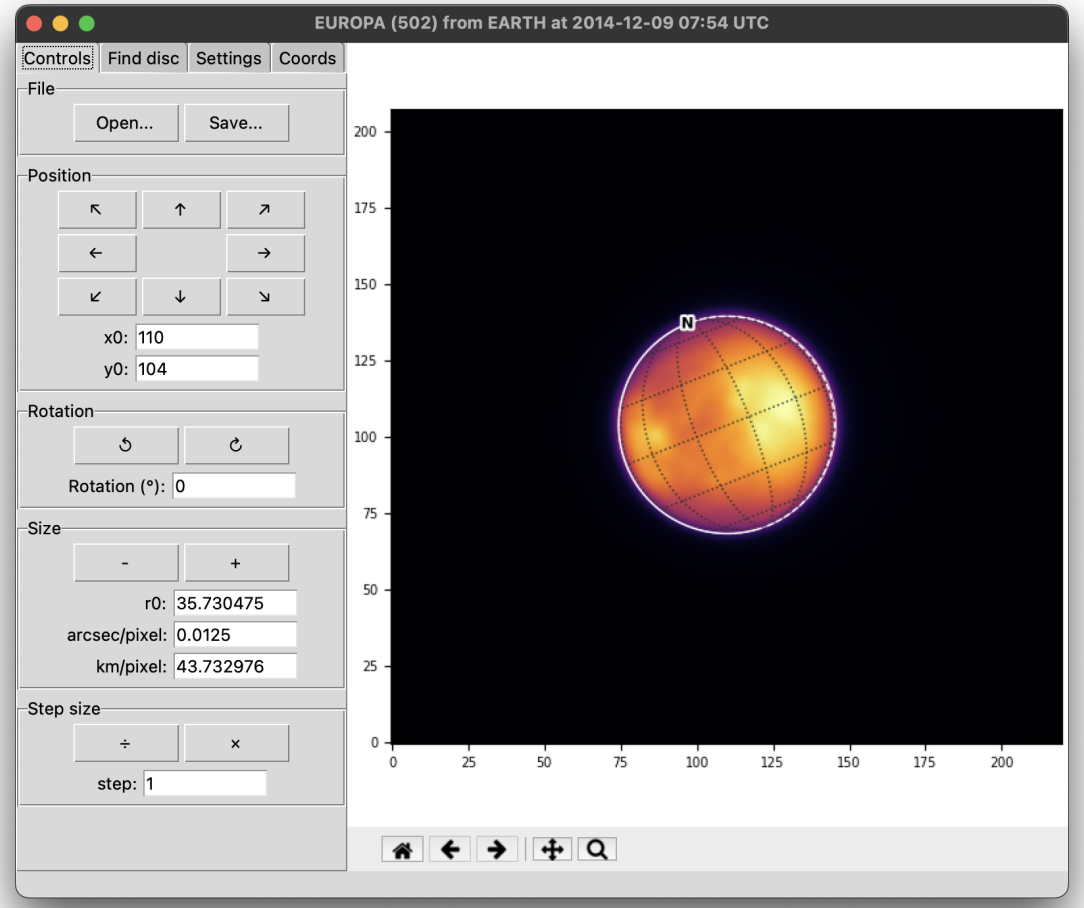
PlanetMapper is an open source Python package for visualising, navigating and mapping Solar System observations.

CHAPTER

ONE

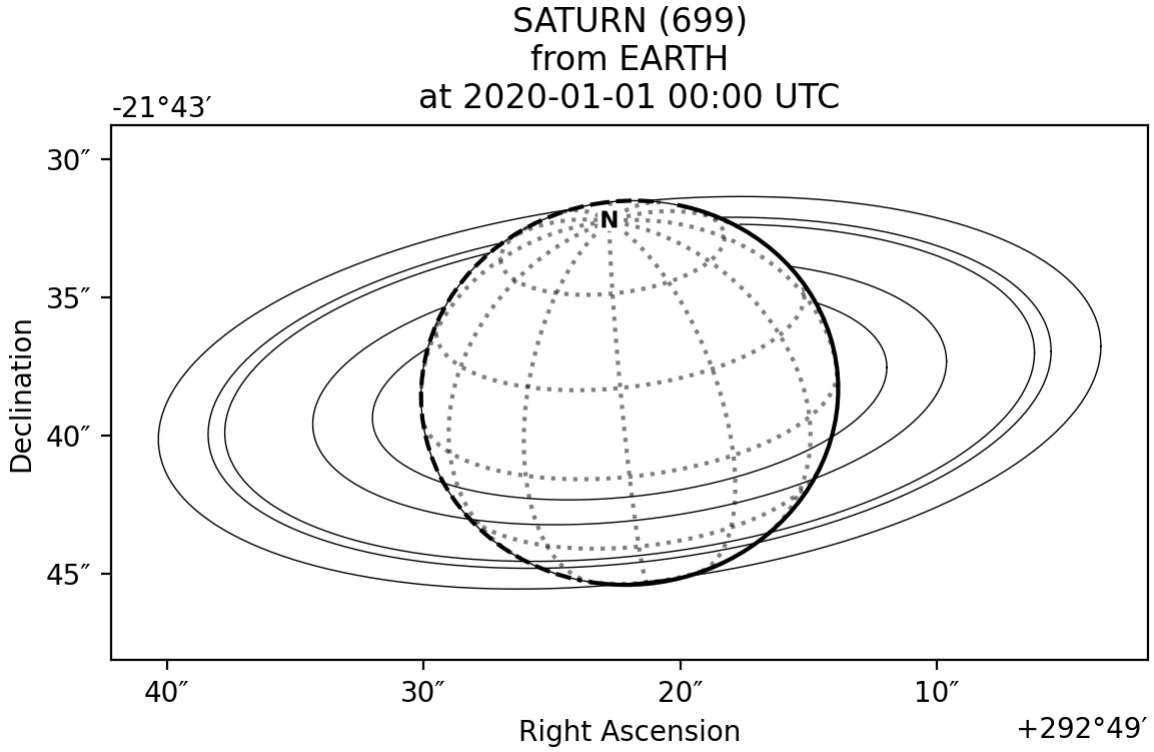
KEY FEATURES

1.1 Fit and map astronomical observations using a full featured user interface

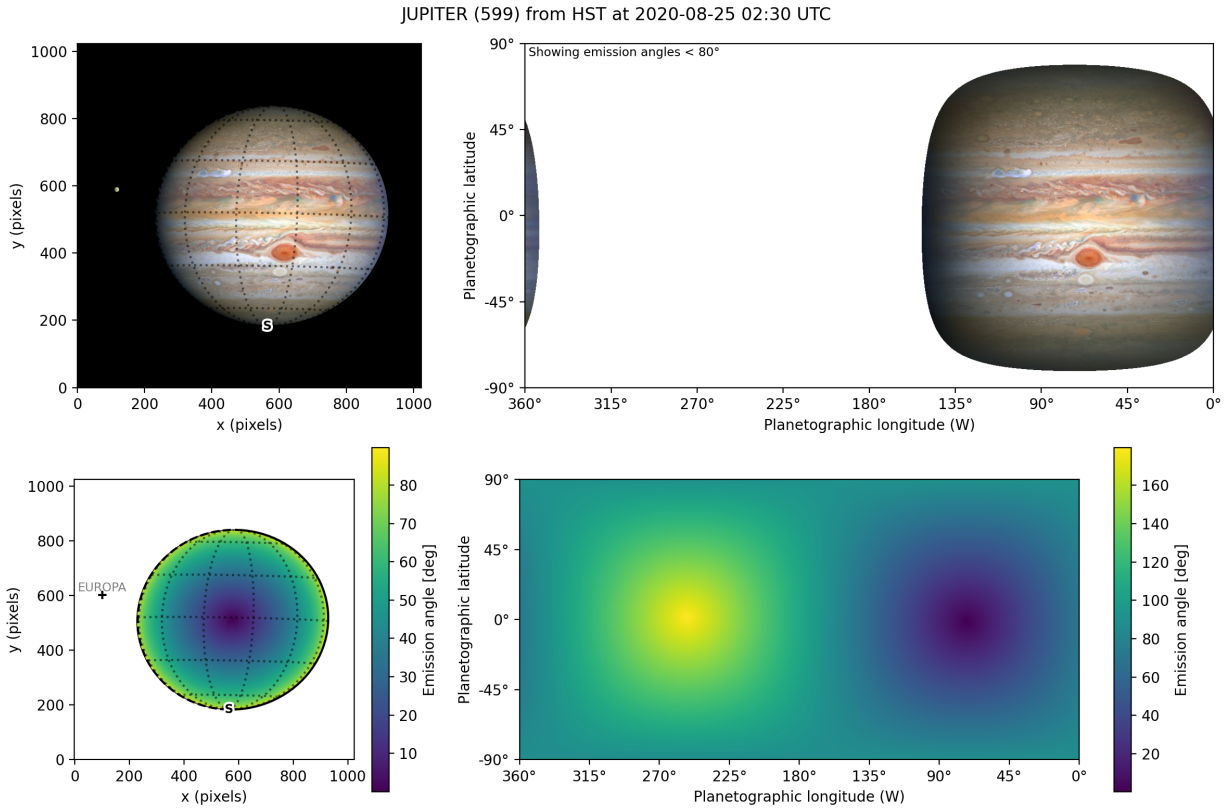


1.2 Easily visualise solar system observations with just a few lines of code

```
body = planetmapper.Body('saturn', '2020-01-01')
body.plot_wireframe_radec()
plt.show()
```



1.3 Convert coordinates, generate backplanes and project maps of telescope observations



CHAPTER

TWO

CITING PLANETMAPPER

If you use PlanetMapper in your research, please cite the following paper:

> King et al., (2023). PlanetMapper: A Python package for visualising, navigating and mapping Solar System observations. Journal of Open Source Software, 8(90), 5728, https://doi.org/10.21105/joss.05728

```
@article{king_2023_planetmapper,
  author  = {King, Oliver R. T. and Fletcher, Leigh N.},
  doi     = {10.21105/joss.05728},
  journal = {Journal of Open Source Software},
  month   = oct,
  number  = {90},
  pages   = {5728},
  title   = {{PlanetMapper: A Python package for visualising, navigating and mapping␣
↪Solar System observations}},
  url     = {https://joss.theoj.org/papers/10.21105/joss.05728},
  volume  = {8},
  year    = {2023}
}
```

2.1 Installation

2.1.1 Installing PlanetMapper

PlanetMapper can easily be installed from PyPI using pip by running:

```
pip install planetmapper
```

or with conda by running:

```
conda install -c conda-forge planetmapper
```

This will automatically install PlanetMapper, along with any dependencies (e.g. NumPy and Astropy) which you do not already have installed. Note that PlanetMapper requires a minimum Python version of 3.10.

2.1.2 Updating PlanetMapper

To upgrade an existing PlanetMapper installation to the latest version, run:

```
pip install planetmapper --upgrade
```

if you installed PlanetMapper with pip, or:

```
conda update planetmapper
```

if you installed PlanetMapper with conda.

The release notes for each version can be found on GitHub, and you can check what version of PlanetMapper you have installed by running:

```
import planetmapper
print(planetmapper.__version__)
```

2.1.3 First steps

The core logic of PlanetMapper uses a series of files called '*SPICE kernels*' which contain the information about the positions and properties of Solar System bodies. Therefore, once you have PlanetMapper installed, you will need to *download the appropriate kernels* before you can properly use PlanetMapper.

Once you have the SPICE kernels downloaded, you can type `planetmapper` in the command line to open an interactive window, or `import planetmapper` in a Python script to get the full functionality.

Hint: Check the *list of common issues* if you encounter any problems when using PlanetMapper

---

2.2 SPICE kernels

Hint: If you are having issues with loading spice kernels after following the guide on this page, check the *list of common issues and solutions*

2.2.1 Introduction

The core logic of PlanetMapper uses the SPICE system, which was developed by NASA's Navigation and Ancillary Information Facility to provide detailed and accurate information about the positions and properties of Solar System bodies and spacecraft. This SPICE database is stored in a series of files called 'SPICE kernels' which must be downloaded for PlanetMapper to function.

Most useful SPICE kernels can be found at https://naif.jpl.nasa.gov/pub/naif/. Each individual SPICE kernel typically contains information about a specific object or set of objects (e.g. one kernel file may contain information about Jupiter and its moons, while another may contain information about a specific spacecraft). Therefore, you only need to download a small subset of the SPICE kernels.

If you already have the appropriate SPICE kernels saved to your computer, you can skip to the *section on customising the kernel directory* below.

2.2.2 Downloading SPICE kernels

To aid in downloading appropriate SPICE kernels, PlanetMapper contains a series of useful functions such as *planetmapper.kernel_downloader.download_urls()* to download kernels from the NAIF database. These functions will automatically download the SPICE kernels to your computer where they can be used by PlanetMapper, so you only need to worry about downloading the kernels once, then PlanetMapper will be able to automatically find and load them in the future.

Note: By default, PlanetMapper will downloaded and search for kernels in a directory named `spice_kernels` within your user directory. If you would like to customise this location (e.g. if you already have kernels saved elsewhere), follow the instructions in the *section on customising the kernel directory* below before downloading any kernels.

Required kernels

To download kernels which are required for virtually every SPICE calculation, run the following commands in Python:

```
from planetmapper.kernel_downloader import download_urls
download_urls('https://naif.jpl.nasa.gov/pub/naif/generic_kernels/lsk/')
download_urls('https://naif.jpl.nasa.gov/pub/naif/generic_kernels/pck/')
```

This will automatically download a series of 'leap second kernels' and 'planetary constant kernels' to your computer. The function *planetmapper.kernel_downloader.download_urls()* effectively replicates the file structure from the NAIF database to your local system. Therefore, the leap second kernels will be automatically downloaded to `~/spice_kernels/naif/generic_kernels/lsk/` and the planetary constant kernels will be downloaded to `~/spice_kernels/naif/generic_kernels/pck/`.

These required kernels are relatively small (~50KB for `.../lsk` and ~50MB for `.../pck`), so downloading them should be relatively fast. If you don't want to use *planetmapper.kernel_downloader.download_urls()*, you can instead manually browse and download files from https://naif.jpl.nasa.gov/pub/naif/ yourself.

Once you have downloaded these required kernels, you will also need to download some of the ephemeris kernels described below.

Planetary ephemeris kernels

Warning: Some SPICE ephemeris kernels can be very large (>1GB), so make sure you have enough free disk space when downloading.

The positions of solar system bodies (e.g. planets and moons) are contained in ephemeris kernels. The kernels are required for each body which you are observing, e.g. if you are using observations of Jupiter you will need to have a Jupiter kernel downloaded.

If you have enough disk space, you can easily download all the planet and moon kernels using:

```
from planetmapper.kernel_downloader import download_urls
download_urls('https://naif.jpl.nasa.gov/pub/naif/generic_kernels/spk/planets/')
download_urls('https://naif.jpl.nasa.gov/pub/naif/generic_kernels/spk/satellites/')
```

These `spk` kernels add up to ~30GB, so if you have limited disk space, you may want to instead download the specific kernels for the bodies you are interested in. Look at the `summaries` and `readme` text files at the top of the planets and satellites archive pages to see which specific files you want to download.

For example, if you are only interested in Jupiter and its moons, you could use:

```
# Note, the exact URLs in this example may not work if new kernel versions are published
from planetmapper.kernel_downloader import download_urls

# Locations of planetary system barycentres:
download_urls('https://naif.jpl.nasa.gov/pub/naif/generic_kernels/spk/planets/de430.bsp')
# Locations of Jupiter and its major satellites:
download_urls('https://naif.jpl.nasa.gov/pub/naif/generic_kernels/spk/satellites/jup365.
↪bsp')

# Optionally download locations of smaller satellites of Jupiter:
download_urls('https://naif.jpl.nasa.gov/pub/naif/generic_kernels/spk/satellites/jup344.
↪bsp')
```

Similarly, if you are interested in Uranus, you could use:

```
# Note, the exact URLs in this example may not work if new kernel versions are published
from planetmapper.kernel_downloader import download_urls

# Locations of planetary system barycentres:
download_urls('https://naif.jpl.nasa.gov/pub/naif/generic_kernels/spk/planets/de430.bsp')
# Locations of Uranus and its major satellites:
download_urls('https://naif.jpl.nasa.gov/pub/naif/generic_kernels/spk/satellites/ura111.
↪bsp')

# Optionally download locations of smaller satellites of Uranus:
download_urls('https://naif.jpl.nasa.gov/pub/naif/generic_kernels/spk/satellites/ura115.
↪bsp')
download_urls('https://naif.jpl.nasa.gov/pub/naif/generic_kernels/spk/satellites/ura116.
↪bsp')
```

Hint: The kernels for the locations of planets are actually located in the generic_kernels/spk/satellites directory, so even if you are only interested in the central planet, you will still need to download at least one kernel from the satellites directory. Search the aa_summaries.txt file for the planet(s) you are interested in to find the required kernel(s).

Spacecraft kernels

If you are using observations from a spacecraft, you will also need to download the ephemeris kernels describing the spacecraft's position over time. For example, if you are using observations from the Hubble Space Telescope, you should run:

```
from planetmapper.kernel_downloader import download_urls
download_urls('https://naif.jpl.nasa.gov/pub/naif/HST/kernels/spk/')
```

The directory name for different missions can be found by searching the NAIF archive.

Other kernels

In some cases, you may require other kernels in addition to those listed above. You should be able to identify the kernels required by searching the NAIF archive. For example, you can download comet ephemerides using

```
from planetmapper.kernel_downloader import download_urls
download_urls('https://naif.jpl.nasa.gov/pub/naif/generic_kernels/spk/comets/')
```

2.2.3 Customising the kernel directory

By default, PlanetMapper downloads and looks for spice kernels in the `~/spice_kernels` directory. However, if needed (e.g. if you already have kernels saved elsewhere), this directory can be customised using the different methods described below. The environment variable method is usually the simplest and easiest.

Method 1: Environment variable

The easiest way to customise the directory is to set the environment variable `PLANETMAPPER_KERNEL_PATH` to point to your desired path. For example, on a Unix-like system, you can add a line to to your `.bash_profile` file to automatically set this environment variable:

```
export PLANETMAPPER_KERNEL_PATH="/path/where/you/save/your/spice/kernels"
```

Method 2: Using `set_kernel_path`

The function *planetmapper.set_kernel_path()* can be used to set the kernel path for a single script. This function *must* be called before using any other `planetmapper` functionality, so it is easiest to run *planetmapper.set_kernel_path()* immediately after importing `planetmapper`:

```
import planetmapper
planetmapper.set_kernel_path('/path/where/you/save/your/spice/kernels')
```

This path should also be set before downloading any SPICE kernels, otherwise they will be downloaded to the incorrect directory:

```
import planetmapper
from planetmapper.kernel_downloader import download_urls
planetmapper.set_kernel_path('/path/where/you/save/your/spice/kernels')

download_urls('https://naif.jpl.nasa.gov/pub/naif/generic_kernels/lsk/')
download_urls('https://naif.jpl.nasa.gov/pub/naif/generic_kernels/pck/')
```

2.2.4 Automatic kernel loading

PlanetMapper will automatically load SPICE kernels the first time any object inheriting from *planetmapper.SpiceBase* (e.g. *planetmapper.Body*) is created. All kernels in the directory returned by *planetmapper.get_kernel_path()* which match any of the patterns `**/*.bsp`, `**/*.tpc` or `**/*.tls` are loaded by default.

If you would like finer control over kernel loading, you can call *planetmapper.base.prevent_kernel_loading()* immediately after importing PlanetMapper to disable automatic kernel loading, then manually load kernels yourself using `spiceypy.furnsh`.

See *planetmapper.SpiceBase* and *planetmapper.SpiceBase.load_spice_kernels()* for more detail about controlling automatic kernel loading.

2.3 Common issues & solutions

Hint: If you find any issues that you cannot solve, please *get in touch through our help pages*!

2.3.1 General solutions

Update PlanetMapper to make sure you are running the latest version. It is possible that your bug may have been fixed in a recent update (you can also check the release notes for each version on GitHub).

2.3.2 Installation issues

PlanetMapper requires a minimum Python version of 3.10, so if you get `ERROR: No matching distribution found for planetmapper` when trying to *install PlanetMapper*, it is likely that your python version is too old. You can check your Python version by running `python3 --version` in a terminal.

2.3.3 `ContentTooShortError` when downloading SPICE kernels

If you get a `ContentTooShortError` when trying to download SPICE kernels, it is likely that the download was interrupted before the file was fully downloaded (e.g. due to an unstable network connection). You can try re-downloading the kernel by running the download command again, or you can manually download the kernel from the NAIF database and place it in the correct directory.

2.3.4 SPICE Errors

If you have any errors caused reported by the SPICE system, it is likely that it doesn't have the correct SPICE kernels loaded. Therefore, make sure you have the *appropriate SPICE kernels downloaded* to your computer and that you have set *the kernel directory* correctly.

`SpiceNOLEAPSECONDS` Error

This error usually occurs when SPICE has not loaded *any* of your desired kernels. This may be because PlanetMapper is not looking in the correct directory for your kernels, so make sure you have set *the kernel directory* correctly.

`SpiceSPKINSUFFDATA` Error

This error usually occurs when SPICE has successfully loaded some kernels, but is missing ephemeris data for either the target body or the observer body. This may be because you have not downloaded the appropriate kernel containing the ephemeris data for this body.

The error message should tell you which body is missing, and you can then identify the correct kernel to download by searching the NAIF database. For example, if you are missing data for a planetary body, you can search the generic_kernels/spk/satellites/aa_summaries.txt file to identify which kernel need downloading.

2.3.5 Planets appear in the wrong position

This is likely to be due to an issue with your SPICE kernels or settings, possible fixes include…

- Make sure you are using the correct observer - e.g. a planet will appear in a different position from Earth and from JWST.
- Make sure you are using the correct observation time - times in PlanetMapper default to UTC, so make sure there are no time zone conversions needed.
- Make sure you have the latest version of any SPICE kernels, especially for any observers like HST or JWST which have have locations which are difficult to predict accurately.
- Make sure you are using the correct aberration correction.
- If you are using WCS information saved in the FITS header to automatically set the disc position, note that telescope pointing information (i.e. the WCS information) is never perfect. For example, due to the errors in guide star tracking, JWST pointing is only accurate to ~0.5".

2.3.6 PlanetMapper crashes when running graphical user interface over SSH

Recent versions of XQuartz appear to have a font handling bug which can cause PlanetMapper to crash when running the user interface over an SSH connection using X11 forwarding:

```
X Error of failed request:  BadValue (integer parameter out of range for operation)
  Major opcode of failed request:  45 (X_OpenFont)
  Value in failed request:  0x60027c
  Serial number of failed request:  3572
  Current serial number in output stream:  3573
```

As a temporary workaround, you can set the `PLANETMAPPER_USE_X11_FONT_BUGFIX` environment variable to `true` on your remote machine before running PlanetMapper if you experience this issue. You can add the following line to to your `.bash_profile` file to automatically set this environment variable:

```
export PLANETMAPPER_USE_X11_FONT_BUGFIX=true
```

This tells the PlanetMapper user interface to replace certain characters with ASCII equivalents (e.g. ↑ is replaced with ^) which seems to prevent the use of the fonts which cause XQuartz to crash. Note that this will make the user interface slightly more ugly, but should not affect functionality. If you are still having issues after trying this workaround, you can add a comment to the GitHub issue.

2.3.7 Wireframe plots appear warped or distorted

This is most likely to occur when using *planetmapper.Body.plot_wireframe_radec()* for a target located near the celestial pole (i.e. the target's declination is near 90° or -90°). The plot can be distorted because spherical coordinates (like RA/Dec) are fundamentally impossible to represent perfectly in a 2D cartesian plot, with the distortion increasing at high declinations near the coordinate singularity at the celestial poles.

To fix this, you can use the *planetmapper.Body.plot_wireframe_angular()*, which by default uses a coordinate system centred on the target body, which minimises any distortion. The origin of the `angular` coordinate system can also be customised to be any point in the sky, for example, using `body.plot_wireframe_angular(origin_ra=0, origin_dec=90)` may be useful for plotting observations in the sky around the north celestial pole.

Plots may also appear distorted if using *planetmapper.Body.plot_wireframe_angular()* with a a custom origin that is a large distance from the target body.

2.3.8 RA/Dec wireframe plots appear split into two halves

If the target body is near RA=0°, the wireframe plot may appear to be split into two halves, due to part of the body having RA values near 0° and part having RA values near 360°. This can be fixed by using `body.plot_wireframe_radec(use_shifted_meridian=True)`, which will plot the wireframe with RA coordinates between -180° and 180°, rather than the default of 0° to 360°.

2.4 Help & support

If you are having any issues with PlanetMapper, check the list of *common issues and their solutions* first. If you can't find a solution there, please get in touch with us and we will do our best to help you out:

- If you've found a bug, please open an issue on GitHub.
- If you have a question or need help, please ask a question on GitHub.
- If you have a have any suggestions, or would otherwise like to get in touch, you can create a new discussion on GitHub.
- If you would like to contribute code to PlanetMapper, see our contributing guidelines on GitHub.

If you are unable to use GitHub, you can also get in contact via email at `ortk2@le.ac.uk`.

2.5 Graphical user interface

The user interface is a convenient and easy way to fit an observation, generate backplanes and map the observed data.

2.5.1 Starting the user interface

The easiest way to run the PlanetMapper user interface is to simply type `planetmapper` in the command line. This will launch a new interactive window where you can choose observation files to open and fit.

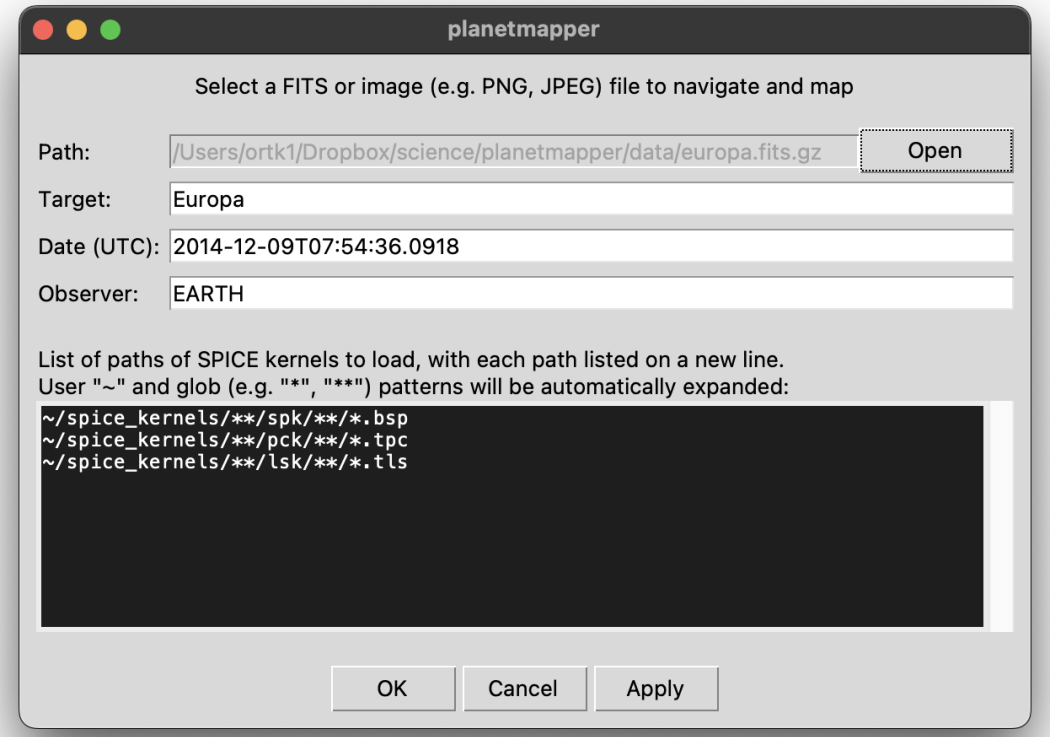
You can create a user interface from a *planetmapper.Observation* object using the *planetmapper.Observation.run_gui()* function. This is mainly useful if you want to combine using a user interface to fit the observation with some Python code to e.g. run some additional analysis. This also gives you access to the full customisation offered by *planetmapper.Observation*, allowing you to specify parameters such as `illumination_source` and `aberration_correction` which cannot be customised using the GUI alone.

It is also possible to start a user interface directly using *planetmapper.gui.GUI.run()*, although the other two methods are generally more useful.

2.5.2 Fitting an observation

Note: You can download the Europa data file used in these examples from the PlanetMapper GitHub repository.

To start, type `planetmapper` into a command line and press `Enter`. This will open a window where you can choose a file to open:[1]

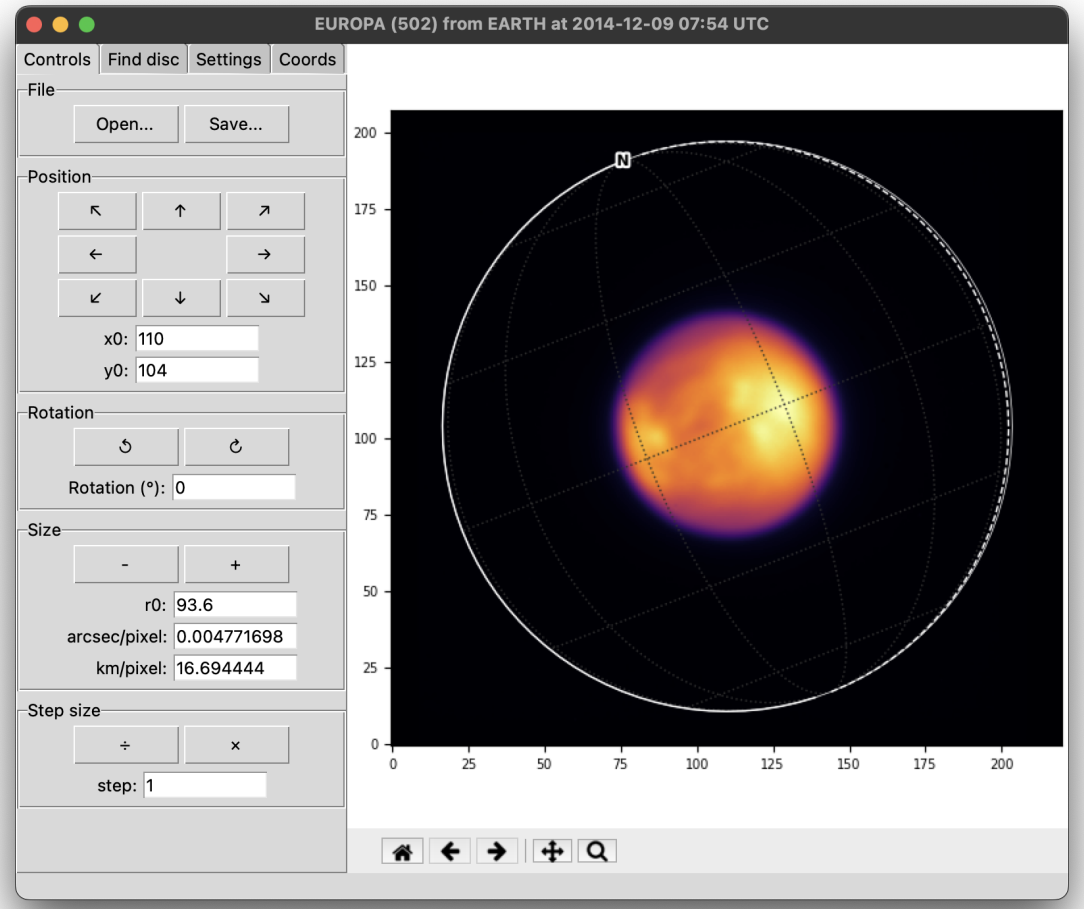


If your data is a FITS file, PlanetMapper will attempt to automatically fill the target, date and observer fields for you with information from the FITS header (but it's worth double checking that the values are what you expect). The date should be in a format which can be understood by SPICE (such as `YYYY-mm-ddTHH:MM:SS`) and should be in UTC. You can also specify a list of *SPICE kernels* to load here - if you're unsure then the default values will probably work.

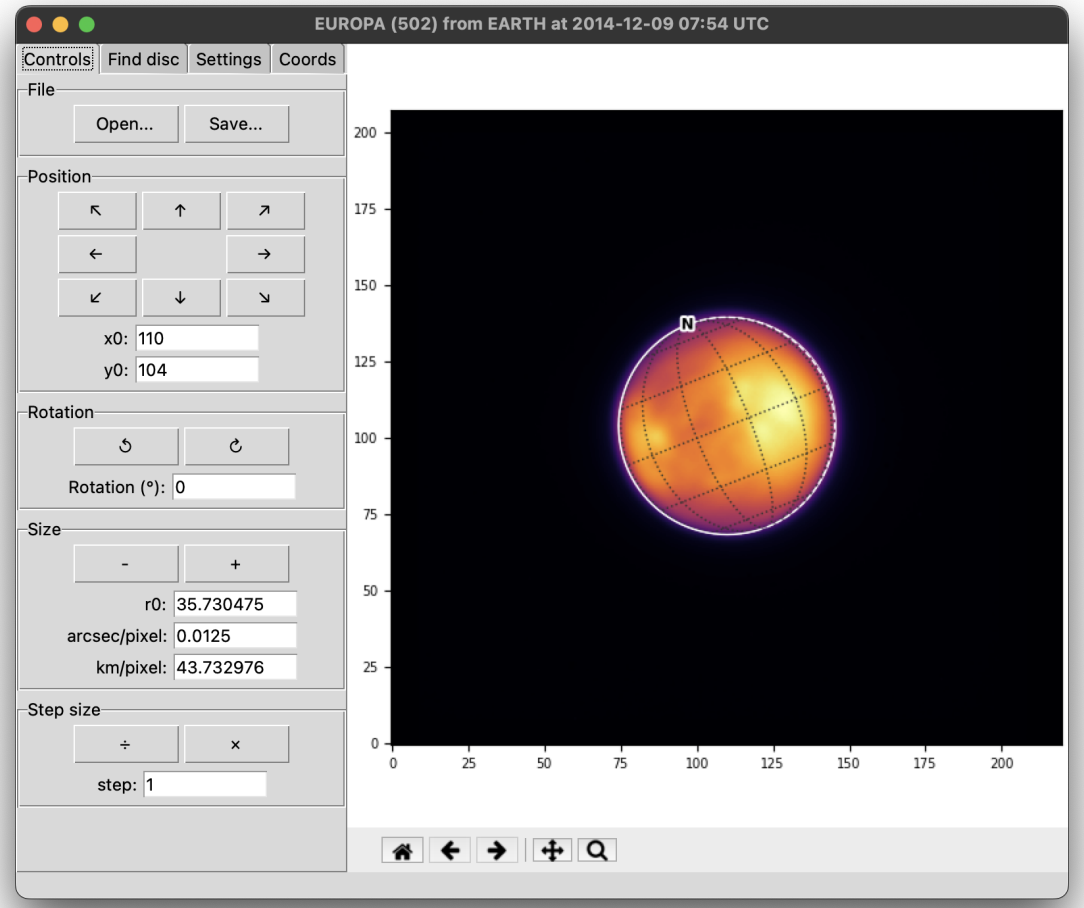
Hint: The target, date and observer fields are passed directly to the `target`, `utc` and `observer` parameters of a *planetmapper.Observation* object, so check the full documentation for *planetmapper.Observation* and *planetmapper.Body* for details of what formats are accepted.

[1] The example Europa dataset is from King et al. (2022). *Compositional Mapping of Europa using MCMC Modelling of Near-IR VLT/SPHERE and Galileo/NIMS Observations.* DOI: 10.3847/PSJ/ac596d.

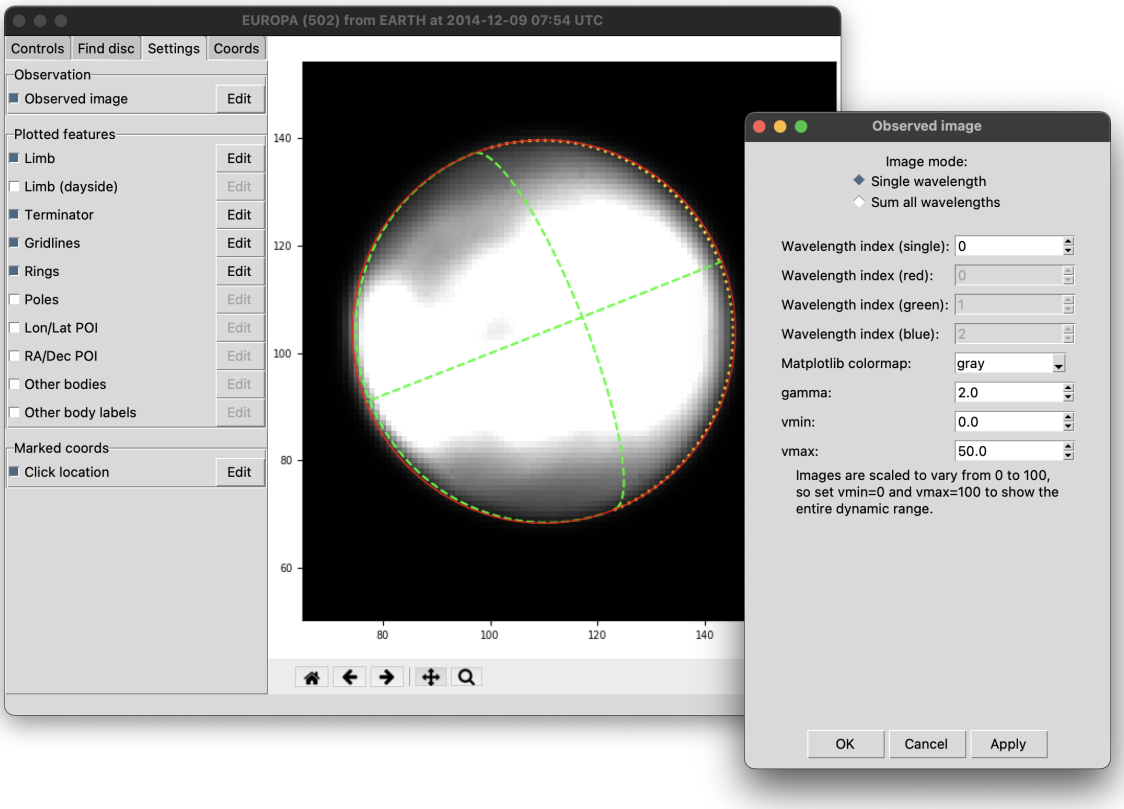
Once you click *OK*, the full fitting window should open. If you get any error messages, then double check the target, date and observer fields for any typos.



This window allows you to fit the observation, so that the fitted disc (the white circle) overlaps nicely with the observed disc. You can use the buttons on the left hand side to move the disc around, or input specific values in the text boxes (for example, you may know the plate scale in arcsec/pixel of the telescope you are using). You can also find the keyboard shortcut for each button by hovering over it and reading the hint at the bottom of the window.



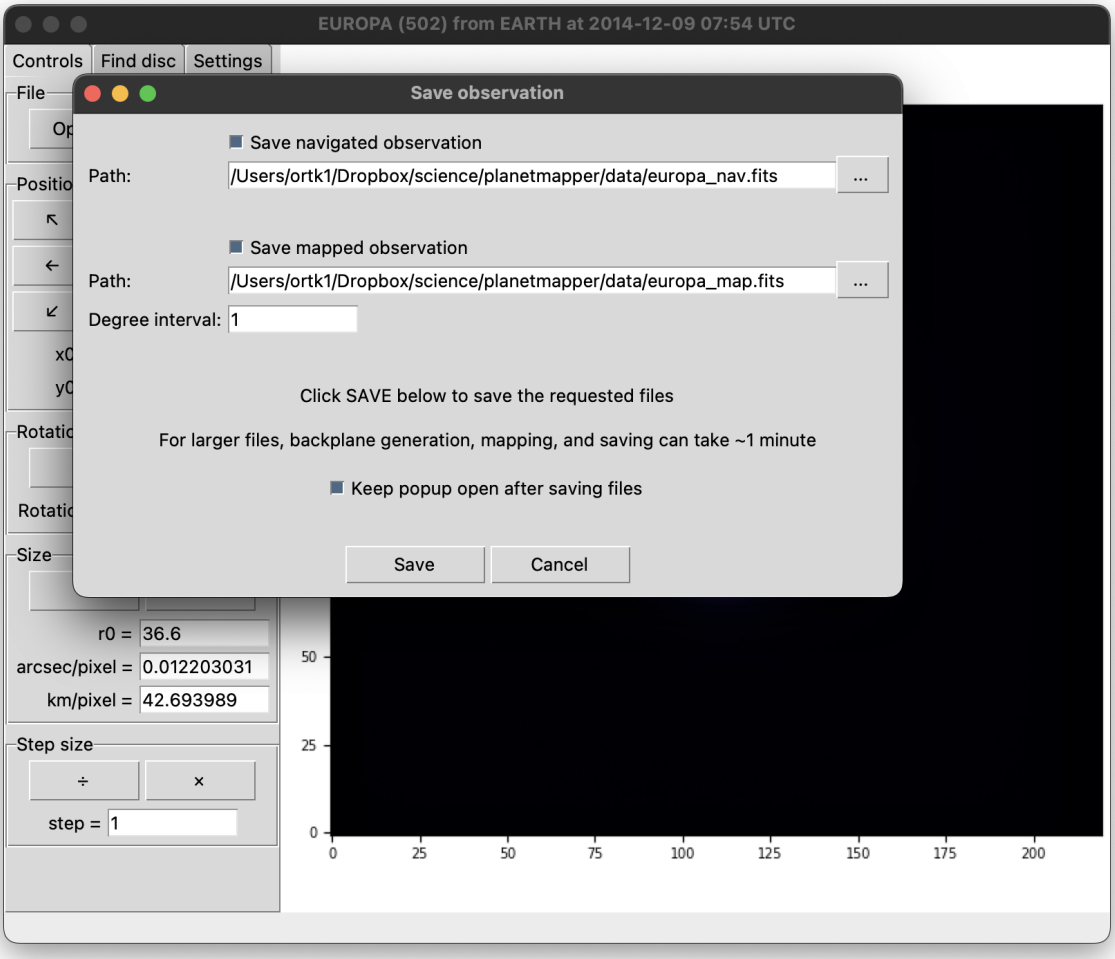
Once the disc is fit, it should look something like the image above. If you want more fine control from clicking the buttons, then you can adjust the step size. It can often be useful to start with a large step size, then decrease it for the final fine alignment.



You can fully customise the appearance of the plot on the right to make fitting easier (or if you just fancy a more exciting colour scheme). In the *Settings* tab, you can toggle the visibility of different plotted elements, and you can click on *Edit* to customise them further. It can be particularly useful to customise the colour scale and brightness of the observed image to increase the contrast around the limb. The zoom and pan buttons beneath the plot can be used to move around the image - click the home button to reset to the default view

You can also use the *Settings* tab to mark points of interest to help with fitting. For example:

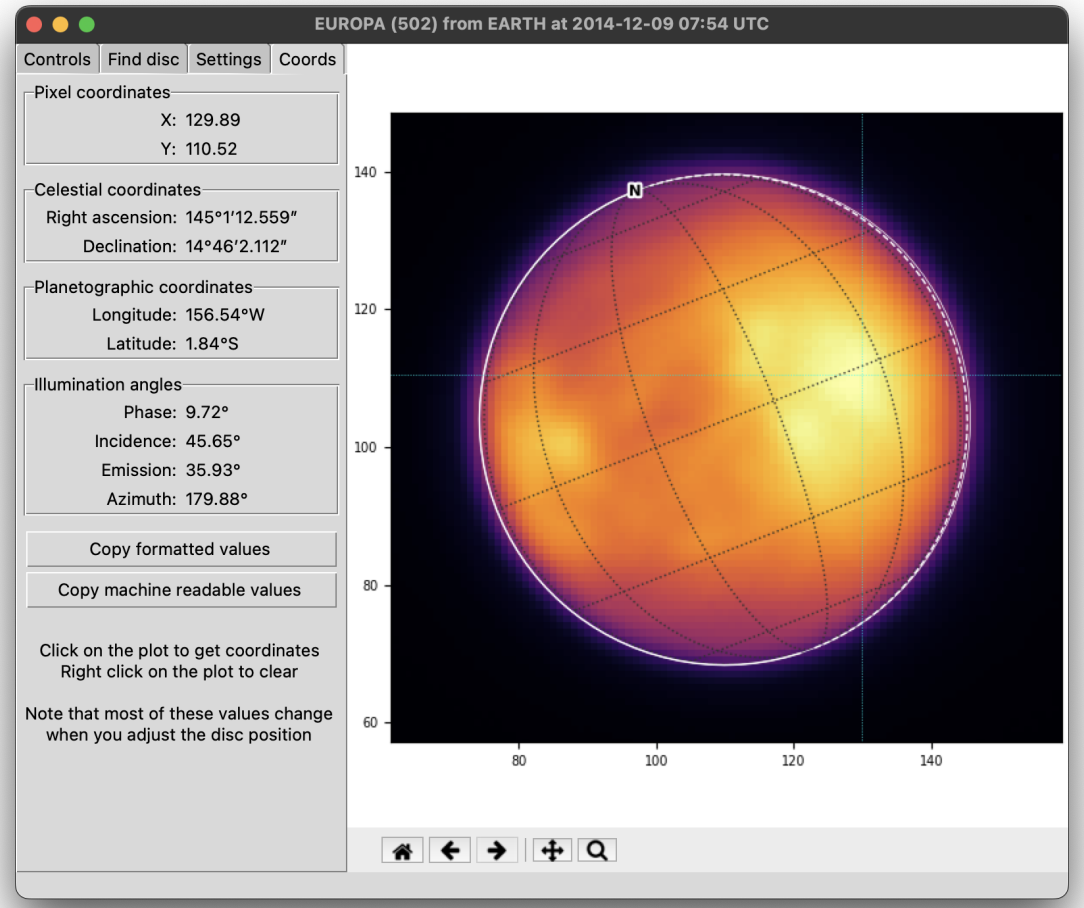
- You can mark a specific location (e.g. a distinctive impact crater) on the surface of the target with a *Lon/Lat POI*.
- You can mark a specific sky coordinate (e.g. a background star) with a *RA/Dec POI*.
- You can mark the location of *Other bodies* (e.g. if you are fitting an observation of Jupiter, you may want to mark the positions of any of its moons which are also in shot).



Once you are happy with the fitting result, click *Save* at the top of the *Controls* tab. This will open a window where you can choose which files to output. You can customise which files to output (with the *Save navigated observation* and *Save mapped observation* checkboxes) and choose the filepath where these files will be saved.

- The navigated observation is similar to the input file, with additional 'FITS backplanes' containing useful information such as the longitude/latitude coordinates for each pixel in the image. This file is generated using the function `planetmapper.Observation.save_observation()`.
- The mapped observation produces a FITS file which contains (as the name suggests…) a mapped version of the observation. This map file will also contain the various useful backplanes. The degree interval option allows you to customise the size of the output map (e.g. degree interval=1 produces a map which is 180x360, degree interval=10 produces a map which is 18x36). This file is generated using the function `planetmapper.Observation.save_mapped_observation()`.

Once you click *Save*, your requested files will be generated and saved. Note that for larger files, this can take around a minute to complete as some of the coordinate conversion calculations are relatively complex.



You can also use the user interface to directly measure the coordinates of points of interest. Simply click on a location in the plot and the coordinate values for that location will be displayed in the *Coords* tab. The coordinate values will also be printed to the command line in a machine readable format that can easily be copied directly into a Python script, JSON database etc. If clicking on the plot isn't updating the coordinates for you, make sure you don't have the pan or zoom buttons selected.

2.5.3 Running the UI from Python

This simple example shows how you could use *planetmapper.Observation.run_gui()* from a Python script to fit multiple observations, then run some custom code on each of them:

```
import glob
import planetmapper

for path in sorted(glob.glob('data/*.fits')):
    # Running from Python allows you to customise SPICE settings like the aberration␣
↪correction
    observation = planetmapper.Observation(path, aberration_correction='CN+S')
```

(continues on next page)

(continued from previous page)

```
    # Run some custom setup
    observation.add_other_bodies_of_interest('Io', 'Europa', 'Ganymede', 'Callisto')
    observation.set_plate_scale_arcsec(42) # set a custom plate scale
    observation.rotation_from_wcs() # get the disc rotation from the header's WCS info

    # Run the GUI to fit the observation interactively
    # This will open a GUI window every loop
    coords = observation.run_gui()

    # More custom code can go here to use the fitted observation...
    # for example, we can print some values for the last click location
    if coords:
        x, y = coords[-1]
        print(observation.xy2lonlat(x, y))
```

2.6 Python package

This page shows some simple examples of using the `planetmapper` package in Python code. For more details, see the full *API documentation*.

For PlanetMapper to function, you will need to download a series of *SPICE kernels* containing the positions and orientations of the solar system bodies you are interested in. The code snippet below will download all the appropriate kernels needed for the examples on this page. For more details about SPICE kernels, including how to choose, download, and use them, see the *SPICE kernel documentation page*.

```
from planetmapper.kernel_downloader import download_urls
# This command will download ~2GB of data
# Note, the exact URLs in this example may not work if new kernel versions are published
download_urls(
    'https://naif.jpl.nasa.gov/pub/naif/generic_kernels/lsk/',
    'https://naif.jpl.nasa.gov/pub/naif/generic_kernels/pck/',
    'https://naif.jpl.nasa.gov/pub/naif/generic_kernels/spk/planets/de430.bsp',
    'https://naif.jpl.nasa.gov/pub/naif/generic_kernels/spk/satellites/jup365.bsp',
    'https://naif.jpl.nasa.gov/pub/naif/generic_kernels/spk/satellites/sat441.bsp',
    'https://naif.jpl.nasa.gov/pub/naif/generic_kernels/spk/satellites/ura111.bsp',
    'https://naif.jpl.nasa.gov/pub/naif/generic_kernels/spk/satellites/nep097.bsp',
    'https://naif.jpl.nasa.gov/pub/naif/HST/kernels/spk/',
)
```

2.6.1 Coordinate conversions

Coordinate conversions can easily be performed using functions such as *`planetmapper.Body.lonlat2radec()`* to calculate the sky coordinates corresponding to a planetographic longitude/latitude coordinate on the surface of the target.

This code shows an example of using some of the functions in *`planetmapper.Body`* to calculate information about observations of Jupiter from Venus:

```
import planetmapper
```

(continues on next page)

(continued from previous page)

```
body = planetmapper.Body('jupiter', '2020-01-01', observer='venus')

coordinates = [(42, 0), (123, 45)]
for lon, lat in coordinates:
    print(f'\nlongitude = {lon}°, latitude = {lat}°')
    if body.test_if_lonlat_visible(lon, lat):
        ra, dec = body.lonlat2radec(lon, lat)
        print(f'  RA = {ra:.4f}°, Dec = {dec:.4f}°')
        if body.test_if_lonlat_illuminated(lon, lat):
            phase, incidence, emission = body.illumination_angles_from_lonlat(lon, lat)
            print(f'  phase angle: {phase:.2f}°')
            print(f'  incidence angle: {phase:.2f}°')
            print(f'  emission angle: {phase:.2f}°')
    else:
        print('  (Not visible)')
```

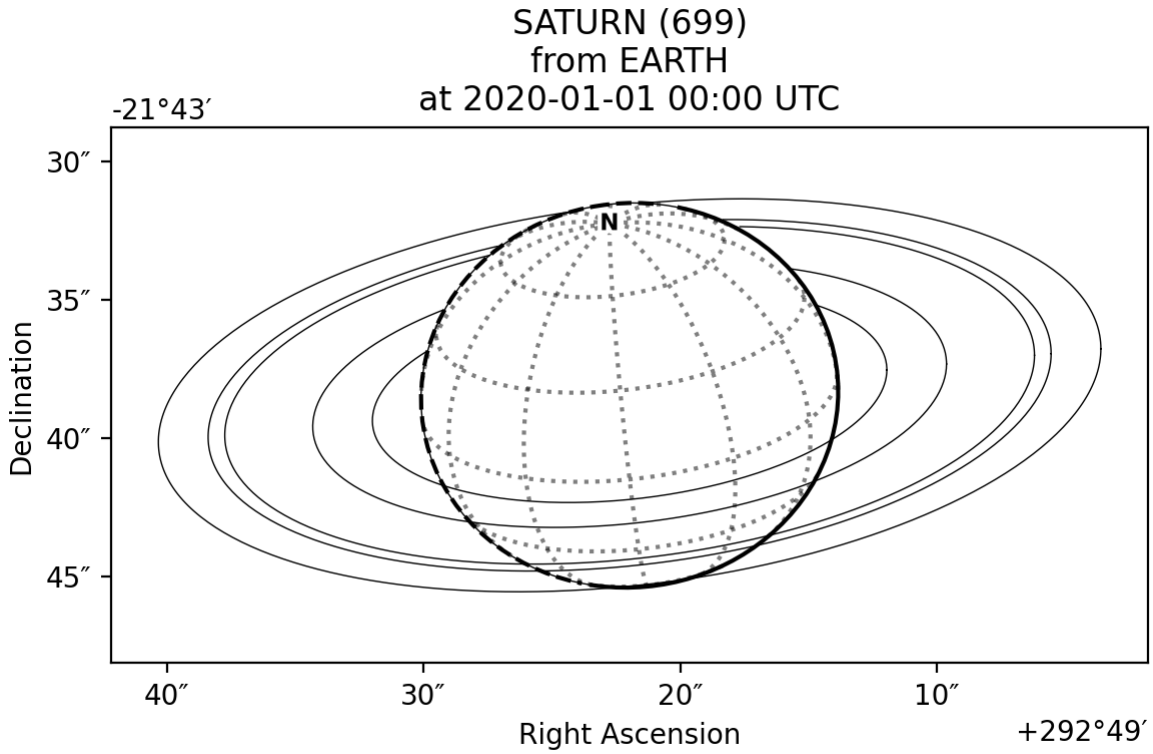
Hint: The main classes in PlanetMapper are subclasses of each other, with *planetmapper.SpiceBase* the parent class of *planetmapper.Body* which is the parent of *planetmapper.BodyXY* which is the parent of *planetmapper.Observation*.

In Python, any functions defined in a parent class are available in any subclasses, so for example, you can use `planetmapper.Observation.lonlat2radec()` exactly the same way as you can use *planetmapper.Body.lonlat2radec()*.

2.6.2 Wireframe plots

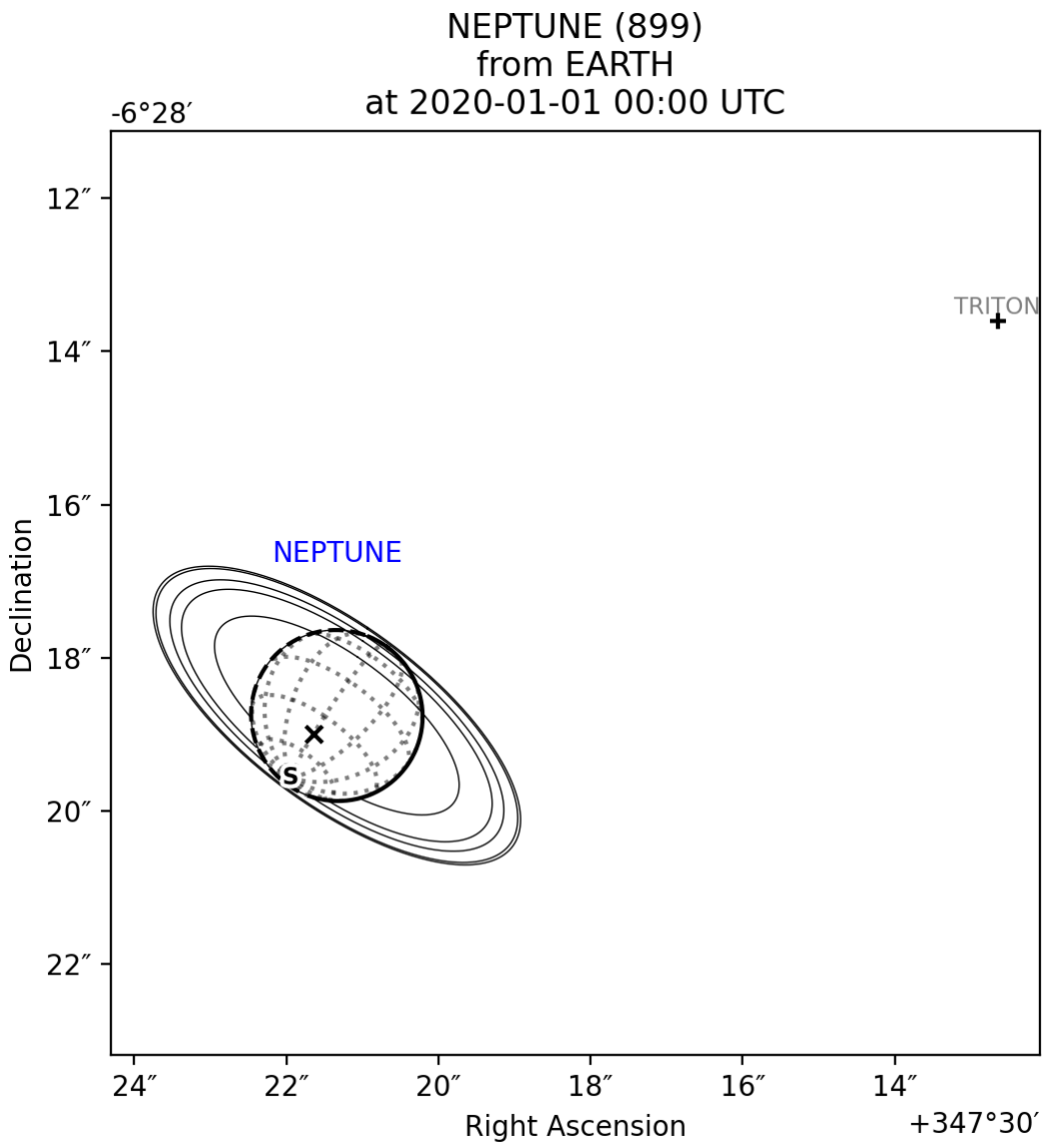
'Wireframe' plots showing the geometry of target bodies can be created quickly and easily using the *planetmapper.Body.plot_wireframe_radec()* command:

```
import planetmapper

body = planetmapper.Body('saturn', '2020-01-01')
body.plot_wireframe_radec(show=True)
```



More complex plots can also be created using the functionality in *planetmapper.Body* and manually adding elements to the plot:

```
import planetmapper
import matplotlib.pyplot as plt

body = planetmapper.Body('neptune', '2020-01-01')

# Add Triton to any wireframe plots
body.add_other_bodies_of_interest('triton')

# Mark this specific coordinate (if visible) on any wireframe plots
body.coordinates_of_interest_lonlat.append((360, -45))

# Add Neptune's rings to the plot
body.add_named_rings()

fig, ax = plt.subplots(figsize=(6, 6), dpi=200)
body.plot_wireframe_radec(ax)

# Manually add some text to the plot
ax.text(
    body.target_ra, body.target_dec + 2 / 60 / 60, 'NEPTUNE', color='b', ha='center'
)

plt.show()
```



Wireframe plot variants

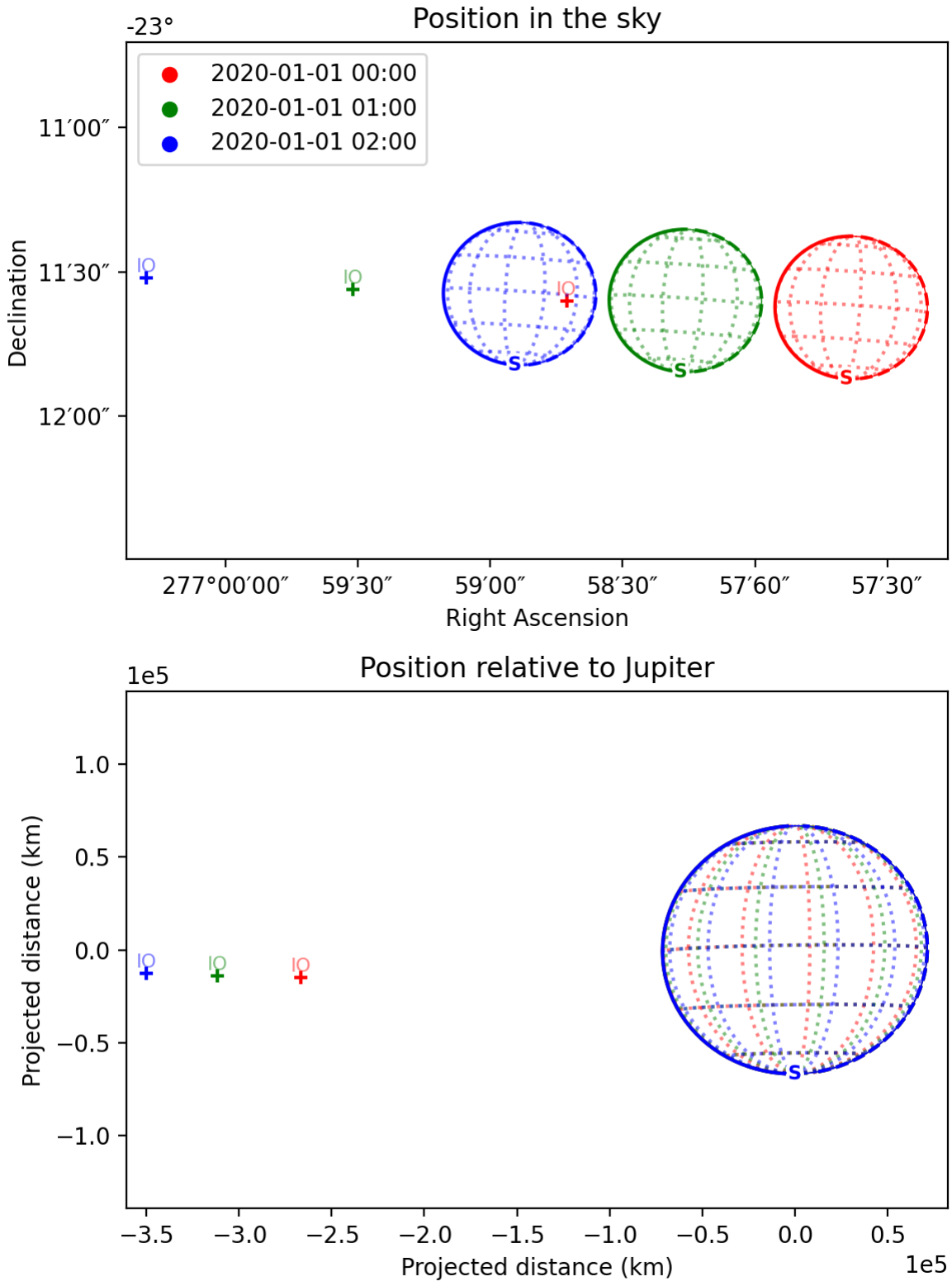
A number of different wireframe plotting options are available:

- *planetmapper.Body.plot_wireframe_radec()* plots in RA/Dec coordinates
- *planetmapper.Body.plot_wireframe_km()* plots in a frame centred showing distances in km from the target body
- *planetmapper.Body.plot_wireframe_angular()* plots in a frame showing angular distances from the the target body
- *planetmapper.BodyXY.plot_wireframe_xy()* plots in image x and y coordinates
- *planetmapper.Body.plot_wireframe_custom()* plots in a custom, user-defined, coordinate system

planetmapper.Body.plot_wireframe_km() is particularly useful for comparing observations taken at different

times, as it standardises the position, orientation and size of the target body. The example below shows multiple observations of Jupiter and Io taken over the space of a few hours. Jupiter moves across the the RA/Dec plot (top), but stays fixed in the km plot (bottom), making it easier to see the relative motion of Io:

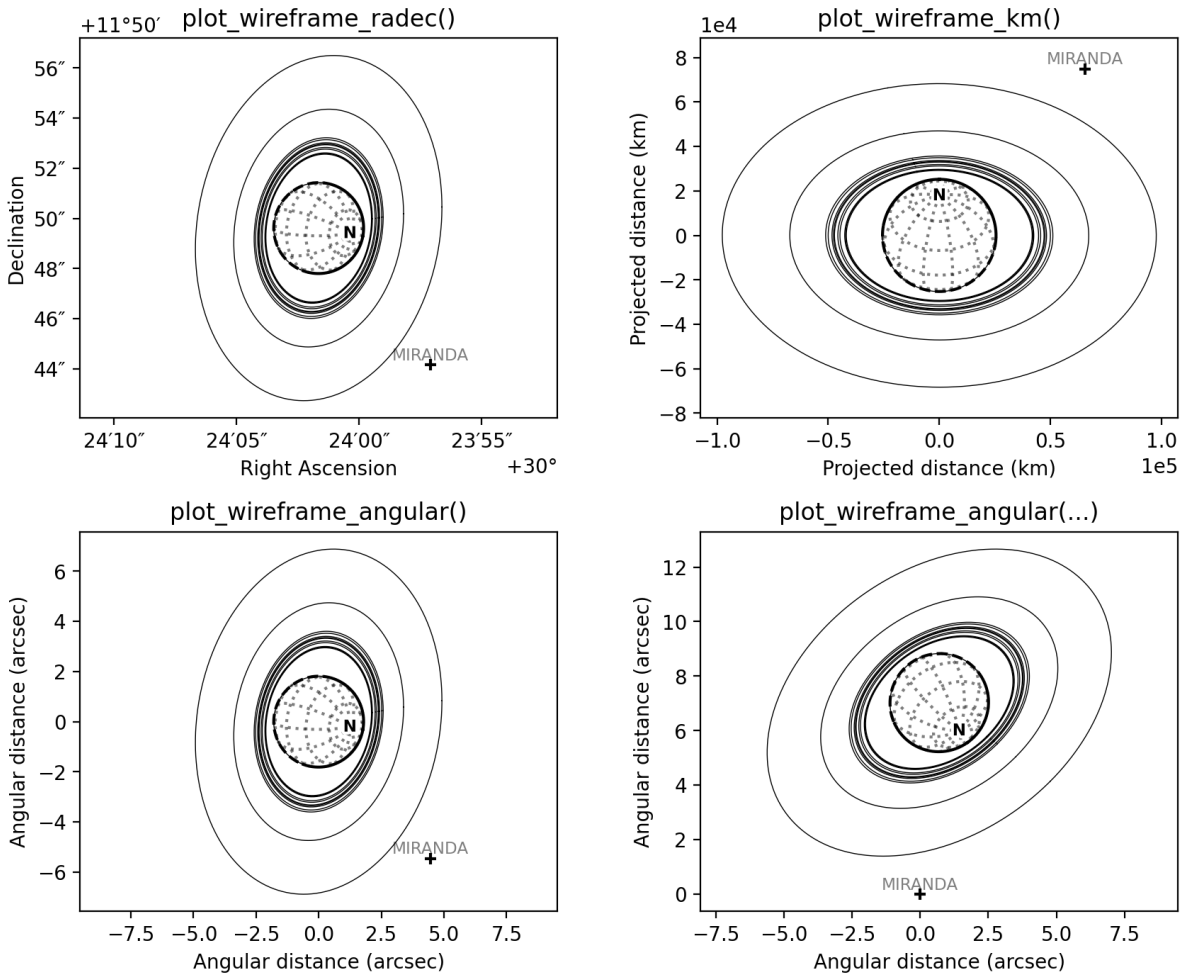
```
import planetmapper
import matplotlib.pyplot as plt
import numpy as np

fig, [ax_radec, ax_km] = plt.subplots(nrows=2, figsize=(6, 8), dpi=200)

dates = ['2020-01-01 00:00', '2020-01-01 01:00', '2020-01-01 02:00']
colors = ['r', 'g', 'b']

for date, c in zip(dates, colors):
    body = planetmapper.Body('jupiter', date)
    body.add_other_bodies_of_interest('Io')
    body.plot_wireframe_radec(ax_radec, color=c)
    body.plot_wireframe_km(ax_km, color=c)

    # Plot some blank data with the correct colour to go on the legend
    ax_radec.scatter(np.nan, np.nan, color=c, label=date)

ax_radec.legend(loc='upper left')

ax_radec.set_title('Position in the sky')
ax_km.set_title('Position relative to Jupiter')

fig.tight_layout()
plt.show()
```



The example below shows how the the same target appears in the `radec`, `km` and `angular` wireframe variants. By default, *`planetmapper.Body.plot_wireframe_angular()`* is centred on the target body, but it can also be customised to have a custom origin and rotation - for example, the fourth plot below is centred on Miranda and rotated

by 45°. In addition to the variants shown here, *planetmapper.BodyXY.plot_wireframe_xy()* is also available for use with *planetmapper.BodyXY* objects to plot in image pixel coordinates (see the Observations section below).

```
import planetmapper
import matplotlib.pyplot as plt

body = planetmapper.Body('uranus', '2020-01-01')
body.add_named_rings()
body.add_other_bodies_of_interest('miranda')

fig, ((ax_radec, ax_km), (ax_angular1, ax_angular2)) = plt.subplots(
    nrows=2,
    ncols=2,
    figsize=(10, 8),
    dpi=200,
    gridspec_kw=dict(hspace=0.3, wspace=0.3),
)

body.plot_wireframe_radec(ax_radec)
ax_radec.set_title('plot_wireframe_radec()')

body.plot_wireframe_km(ax_km)
ax_km.set_title('plot_wireframe_km()')

body.plot_wireframe_angular(ax_angular1)
ax_angular1.set_title('plot_wireframe_angular()')

miranda = body.create_other_body('miranda')
# angular plot centred on custom RA/Dec and with a custom rotation
body.plot_wireframe_angular(
    ax_angular2,
    origin_ra=miranda.target_ra,
    origin_dec=miranda.target_dec,
    coordinate_rotation=-45,
)
ax_angular2.set_title('plot_wireframe_angular(...)')

plt.show()
```



Customising wireframe plots

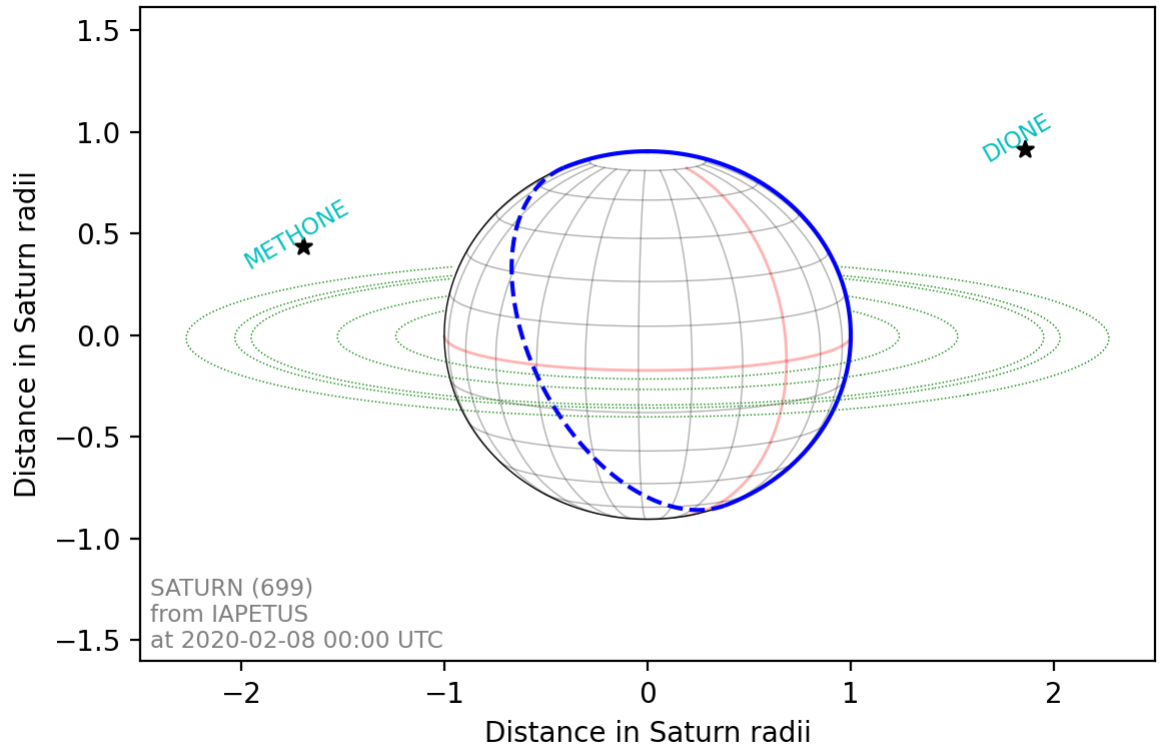
The appearance and units of wireframe plots can be fully customised to suit your needs. For example, the code below shows how the `scale_factor` argument is used to customise the coordinate units, and the `formatting` argument is used to pass arguments to matplotlib when plotting the individual elements of the wireframe. See *planetmapper.Body.plot_wireframe_radec()* for more details on formatting individual plots, and changing the default formatting for all wireframe plots.

```
import planetmapper
import matplotlib.pyplot as plt

body = planetmapper.Body('saturn', '2020-02-08', observer='iapetus')
body.add_other_bodies_of_interest('dione', 'methone')
body.plot_wireframe_km(
    ax,
    scale_factor=1 / body.r_eq,  # use units of Saturn radii rather than km
    add_title=False,
    label_poles=False,
    indicate_equator=True,
    indicate_prime_meridian=True,
```

(continues on next page)

(continued from previous page)

```
    grid_interval=15,
    grid_lat_limit=75,
    formatting={
        'grid': {'linestyle': '-', 'linewidth': 0.5, 'alpha': 0.3},
        'prime_meridian': {'linewidth': 1, 'color': 'r'},
        'equator': {'linewidth': 1, 'color': 'r'},
        'terminator': {'color': 'b'},
        'limb_illuminated': {'color': 'b'},
        'ring': {'color': 'g', 'linestyle': ':'},
        'other_body_of_interest_marker': {'marker': '*'},
        'other_body_of_interest_label': {'color': 'c', 'rotation': 30, 'alpha': 1},
    },
)
ax.set_xlabel('Distance in Saturn radii')
ax.set_ylabel('Distance in Saturn radii')
ax.annotate(
    body.get_description(),
    (0.01, 0.02),
    xycoords='axes fraction',
    color='0.5',
    size='small',
)
plt.show()
```

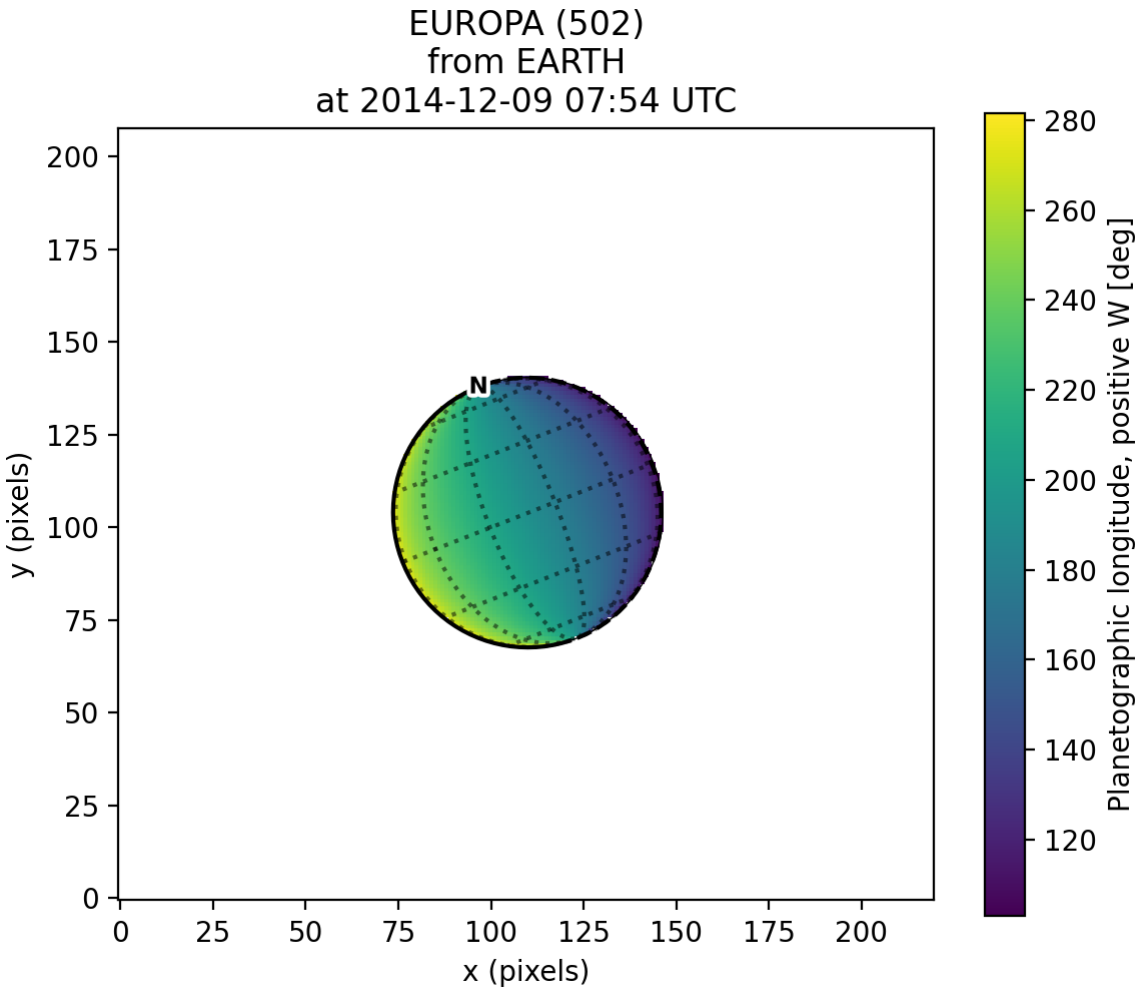


2.6.3 Observations, backplanes and mapping

Note: You can download an example Europa data file from the PlanetMapper GitHub repository.

planetmapper.Observation objects can be created to calculate information about a specific observation. If the observed data is saved in a FITS file with appropriate header information, a *planetmapper.Observation* object can be created using only the path to that file - target, date and observer information can all be derived automatically from the header. The example below creates an Observation object, and uses it to plot an image containing showing the longitude value of each pixel:

```
import planetmapper
import matplotlib.pyplot as plt

observation = planetmapper.Observation('europa.fits')

# Set the disc position
observation.set_plate_scale_arcsec(12.25e-3)
observation.set_disc_params(x0=110, y0=104)

observation.plot_backplane_img('LON-GRAPHIC')
plt.show()
```



A range of backplane images can be generated - see *Default backplanes* for a list of the backplanes available by default. These backplanes can be saved to a FITS file for future use using *planetmapper.Observation.save_observation()*. A mapped version of the image and backplanes can likewise be saved using *planetmapper.Observation.save_mapped_observation()*:

```
import planetmapper

observation = planetmapper.Observation('europa.fits')

# Set the disc position
observation.set_plate_scale_arcsec(12.25e-3)
observation.set_disc_params(x0=110, y0=104)

observation.save_observation('europa_navigated.fits')
observation.save_mapped_observation('europa_mapped.fits')
```

Mapped data can also be manipulated and plotted directly. In the example below, we use *planetmapper.Observation.get_mapped_data()* and *planetmapper.BodyXY.get_backplane_map()* to directly access, manipulate and plot the mapped data and backplanes:[1]

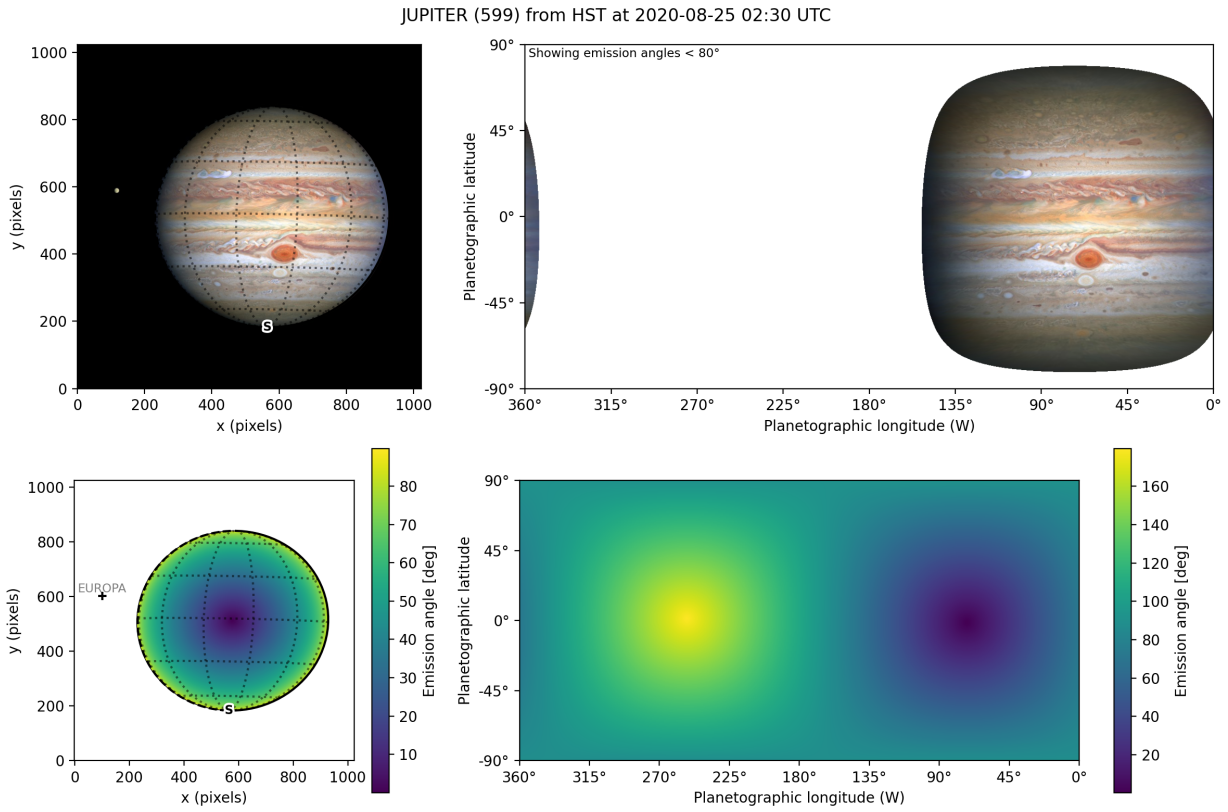
[1] The Jupiter image is from the OPAL program using the Hubble Space Telescope. Credit: *NASA, ESA, STScI, A. Simon (Goddard Space Flight Center), and M.H. Wong (University of California, Berkeley) and the OPAL team*

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

# This uses a JPG image, so we need to manually specify details (e.g. target)
observation = planetmapper.Observation(
    'jupiter.jpg',
    target='jupiter',
    utc='2020-08-25 02:30:40',
    observer='HST',
    show_progress=True, # show progress bars for slower functions
)

# Run the GUI to fit the disc interactively
observation.run_gui()

fig, axs = plt.subplots(
    nrows=2, ncols=2, figsize=(12, 8), dpi=200, width_ratios=[1, 2]
)

# Do a nice RGB plot of the data in the top left
rgb_img = np.moveaxis(observation.data, 0, 2)  # imshow needs wavelength index last
axs[0, 0].imshow(rgb_img, origin='lower')
observation.plot_wireframe_xy(axs[0, 0])

# Plot the emission angle backplane in the bottom left
observation.add_other_bodies_of_interest('Europa')  # mark Europa on this plot
observation.plot_backplane_img('EMISSION', ax=axs[1, 0])

# Plot the mapped emission angle backplane in the bottom right
observation.plot_backplane_map('EMISSION', ax=axs[1, 1])


# Plot a mapped RGB image of the data in the top right
degree_interval = 0.25  # Plot maps with 4 pixels/degree
emission_cutoff = 80

mapped_data = observation.get_mapped_data(degree_interval)  # get the mapped data
rgb_map = np.moveaxis(mapped_data, 0, 2)  # imshow needs wavelength index last
rgb_map = planetmapper.utils.normalise(rgb_map)  # normalise to make plot look nicer

# Only plot areas with emission angles <80deg
emission_map = observation.get_backplane_map('EMISSION', degree_interval)
for idx in range(3):
    rgb_map[:, :, idx][np.where(emission_map > emission_cutoff)] = 1

# Display mapped image and add a useful annotation
observation.imshow_map(rgb_map, ax=axs[0, 1])
axs[0, 1].annotate(
    f'Showing emission angles < {emission_cutoff}°',
    (0.005, 0.99),
    xycoords='axes fraction',
    size='small',
```

(continues on next page)

(continued from previous page)

```
    va='top',
)


# Add some general formatting
for ax in axs.ravel():
    ax.set_title('')
fig.suptitle(observation.get_description(multiline=False))
fig.tight_layout()

plt.show()
```



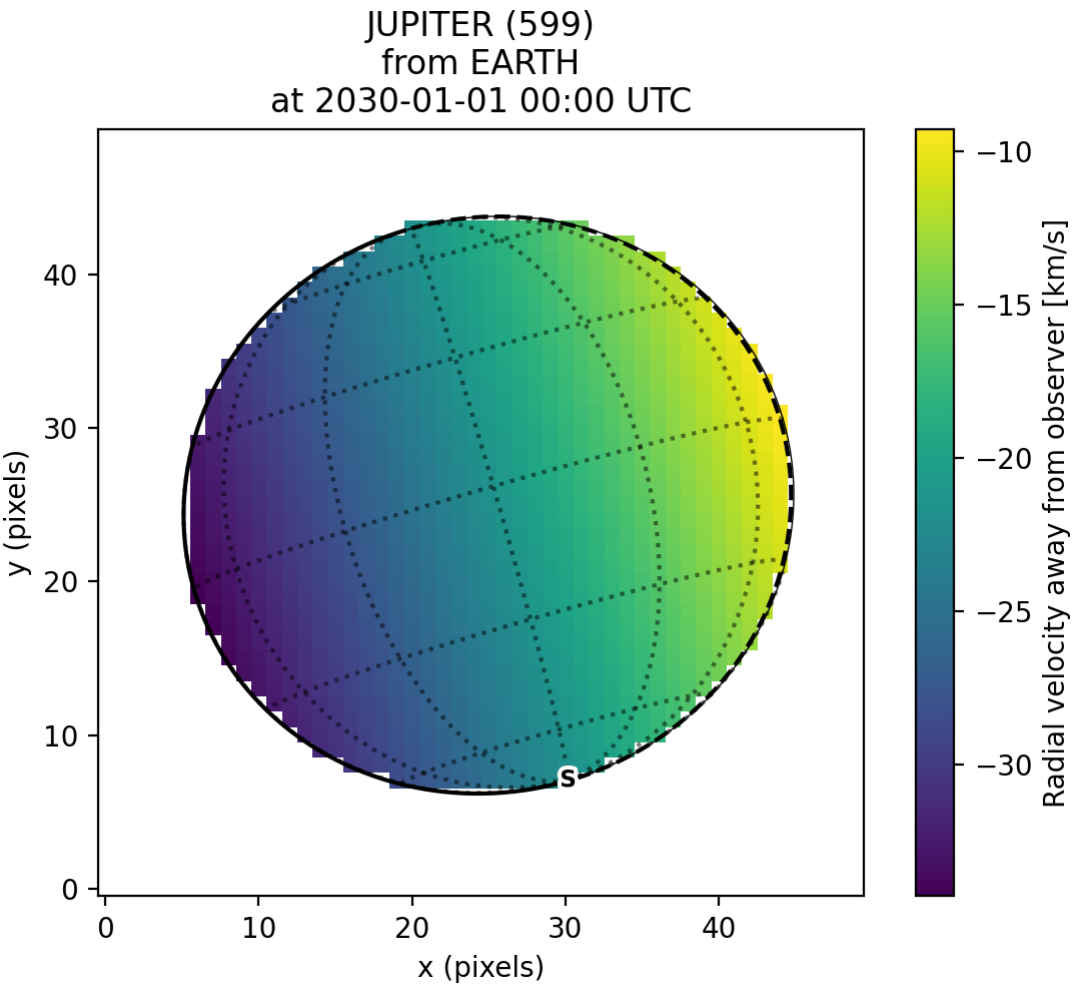
Backplanes can also be generated for observations which do not exist using *planetmapper.BodyXY*:

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

# Create an object representing how Jupiter would appear in a 50x50 pixel image
# taken from Earth at a specific time
body = planetmapper.BodyXY('jupiter', utc='2030-01-01', observer='Earth', sz=50)
body.set_disc_params(x0=25, y0=25, r0=20)

fig, ax = plt.subplots(figsize=(6, 5), dpi=200)
body.plot_backplane_img('RADIAL-VELOCITY', ax=ax)
```

(continues on next page)

(continued from previous page)

```
fig.tight_layout()
plt.show()

# Backplane images can also be accessed and manipulated directly
radial_velocities = body.get_backplane_img('RADIAL-VELOCITY')
print(f'Average radial velocity: {np.nanmean(radial_velocities):.2f} km/s')

# Average radial velocity: -21.78 km/s
```



2.6.4 Cache behaviour

The generation of backplanes and projected mapped data can be slow for larger datasets. Therefore, *planetmapper.BodyXY* and *planetmapper.Observation* objects automatically cache the results of various expensive function calls so that they do not have to be recalculated. This cache management happens automatically behind the scenes, so you should never have to worry about dealing with it directly. For example, when any disc parameters are changed, the cache is automatically cleared as the cached results will no longer be valid.

```
import planetmapper
```

(continues on next page)

(continued from previous page)

```
# Create a new object
body = planetmapper.BodyXY('Jupiter', '2000-01-01', sz=500)
body.set_disc_params(x0=250, y0=250, r0=200)
# At this point, the cache is completely empty

# The intermediate results used in generating the incidence angle backplane
# are cached, speeding up any future calculations which use these
# intermediate results:
body.get_backplane_img('INCIDENCE') # Takes ~10s to execute
body.get_backplane_img('INCIDENCE') # Executes instantly
body.get_backplane_img('EMISSION') # Executes instantly

# When any of the disc parameters are changed, the xy <-> radec conversion
# changes so the cache is automatically cleared (as the cached intermediate
# results are no longer valid):
body.set_r0(190) # This automatically clears the cache
body.get_backplane_img('EMISSION') # Takes ~10s to execute
body.get_backplane_img('INCIDENCE') # Executes instantly
```

The methods which cache their results include…

- *planetmapper.BodyXY.get_backplane_img()*
- *planetmapper.BodyXY.get_backplane_map()*
- *planetmapper.Observation.get_mapped_data()*
- *planetmapper.Observation.save_observation()* and equivalent option in the GUI
- *planetmapper.Observation.save_mapped_observation()* and equivalent option in the GUI

Note: The Python script used to generate all the figures shown on this page can be found here

2.7 planetmapper

Hint: See also the *page of examples* of using the PlanetMapper Python package

PlanetMapper: A Python package for visualising, navigating and mapping Solar System observations.

The core logic of this code is based on conversion between different coordinate systems of interest. The `xy` and `radec` coordinate systems define positions from the point of view of the observer while the `lonlat` coordinate system defines locations on the surface of the target body:

`xy`: image pixel coordinates. These coordinates count the number of pixels in an observed image with the bottom left pixel defined as `(0, 0)`, and the `x` and `y` coordinates defined as normal. Integer coordinates represent the middle of the corresponding pixel, so `(0, 3)` covers `x` values from -0.5 to 0.5 and `y` values from 2.5 to 3.5.

`radec`: observer frame RA/Dec sky coordinates. These are the right ascension and declination which define the position in the sky of a point from the point of view of the observer. These coordinates are expressed in degrees. See Wikipedia for more details.

`lonlat`: planetographic coordinates on target body. These are the planetographic longitude and latitude coordinates of a point on the surface of the target body. These coordinates are expressed in degrees. See Wikipedia and the SPICE documentation for more details.

`km`: defines the distance in the image plane from the centre of the target body in km with the target's north pole pointing up. This coordinate system is similar to the `radec` and `xy` coordinate systems, but has the image zoomed so that the planet's radius is fixed and rotated so that the north pole points up. It can therefore be useful for comparing observations of the same target taken at different times.

`angular`: relative angular coordinates in arcseconds. By default, the angular coordinates are centred on the target body, with the same rotation as the `radec` coordinates, meaning the angular coordinates define the distance in arcseconds from the centre of the target body. However, the origin and rotation of the angular coordinates can also be customised to measure the angular distance in arcseconds relative to an arbitrary point in the sky. See *`Body.radec2angular()`* for more details on customising the `angular` coordinates. Similarly to the `km` coordinate system, this can be useful for comparing observations of the same target taken at different times. It also can be used to minimise the distortion present when plotting `radec` coordinates for targets located near the celestial pole.

Dimension	Unit
Angles (RA, Dec, longitude, latitude…)	degrees
Angles (relative angular coordinates, plate scale…)	arcseconds[1]
Distances	km
Time intervals	seconds
Speeds	km/s
Dates (timezone)	UTC

Note: By default, all angles should be degrees unless using a function/value explicitly named with `_arcsec` or `_radians`, or using the relative `angular` coordinate system. Note that angles in SPICE are radians, so extra care should be taken converting to/from SPICE values.

These additional coordinate systems are mainly used for internal calculations and to interface with SPICE:

- `targvec` - target frame rectangular vector.
- `obsvec` - observer frame (e.g. J2000) rectangular vector.
- `obsvec_norm` - normalised observer frame rectangular vector.
- `rayvec` - target frame rectangular vector from observer to point.

This code makes extensive use of the the `spiceypy` package which provides a Python wrapper around NASA's `cspice` toolkit. See the spiceypy documentation and the SPICE documentation for more information.

If you use PlanetMapper in your research, please *cite the following paper*:

> King et al., (2023). PlanetMapper: A Python package for visualising, navigating and mapping Solar System observations. Journal of Open Source Software, 8(90), 5728, https://doi.org/10.21105/joss.05728

Warning: This code is in active development, so may contain bugs! Any issues, bugs and suggestions can be reported on GitHub.

`planetmapper.set_kernel_path`(*path: str | os.PathLike | None*) → None

Set the path of the directory containing SPICE kernels. See *the kernel directory documentation* for more detail.

[1] 3600 arcseconds = 1 degree

Parameters

path – Directory which PlanetMapper will search for SPICE kernels. If `None`, then the default value of `'~/spice_kernels/'` will be used.

`planetmapper.`**get_kernel_path**(*return_source: bool = False*) → str | tuple[str, str]

Get the path of the directory of SPICE kernels used in PlanetMapper.

1. If a kernel path has been manually set using *set_kernel_path()*, then this path is used.
2. Otherwise the value of the environment variable `PLANETMAPPER_KERNEL_PATH` is used.
3. If `PLANETMAPPER_KERNEL_PATH` is not set, then the default value, `'~/spice_kernels/'` is used.

Parameters

return_source – If `True`, return a tuple of the kernel path and a string which indicates the source of the kernel path. If `False` (the default), return only the kernel path. The possible source strings are: `'set_kernel_path()'`, `'PLANETMAPPER_KERNEL_PATH'`, and `'default'`.

Returns

The path of the directory of SPICE kernels used in PlanetMapper. If `return_source` is `True`, then a tuple of the kernel path and a string indicating the source of the kernel path is returned.

class `planetmapper.`**SpiceBase**(*show_progress: bool = False*, *optimize_speed: bool = True*, *auto_load_kernels: bool = True*, *kernel_path: str | None = None*, *manual_kernels: None | list[str] = None*)

Bases: `object`

Class containing methods to interface with spice and manipulate coordinates.

This is the base class for all the main classes used in planetmapper.

Parameters

- **show_progress** – Show progress bars for long running processes. This is mainly useful for complex functions in derived classes, such as backplane generation in *BodyXY*. These progress bars can be quite messy, but can be useful to keep track of very long operations.
- **optimize_speed** – Toggle speed optimizations. For typical observations, the optimizations can make code significantly faster with no effect on accuracy, so should generally be left enabled.
- **auto_load_kernels** – Toggle automatic kernel loading with *load_spice_kernels()*.
- **kernel_path** – Passed to *load_spice_kernels()* if `load_kernels` is True. It is recommended to use *set_kernel_path()* instead of passing this argument.
- **manual_kernels** – Passed to *load_spice_kernels()* if `load_kernels` is True. It is recommended to use *planetmapper.base.prevent_kernel_loading()* then manually load kernels yourself instead of passing this argument.

copy() → Self

Return a copy of this object.

standardise_body_name(*name: str | int*) → str

Return a standardised version of the name of a SPICE body.

This converts the provided `name` into the SPICE ID code with `spice.bods2c`, then back into a string with `spice.bodc2s`. This standardises to the version of the name preferred by SPICE. For example, `'jupiter'`, `'JuPiTeR'`, `' Jupiter '`, `'599'` and `599` are all standardised to `'JUPITER'`

Parameters
name – The name of a body (e.g. a planet). This can also be the numeric ID code of a body.

Returns
Standardised version of the body's name preferred by SPICE.

Raises
NotFoundError – If SPICE does not recognise the provided name

et2dtm(*et: float*) → datetime

Convert ephemeris time to a Python datetime object.

Parameters
et – Ephemeris time in seconds past J2000.

Returns
Timezone aware (UTC) datetime corresponding to et.

static mjd2dtm(*mjd: float*) → datetime

Convert Modified Julian Date into a python datetime object.

Parameters
mjd – Float representing MJD.

Returns
Python datetime object corresponding to mjd. This datetime is timezone aware and set to the UTC timezone.

speed_of_light() → float

Return the speed of light in km/s. This is a convenience function to call spice.clight().

Returns
Speed of light in km/s.

calculate_doppler_factor(*radial_velocity: Numeric*) → Numeric

Calculates the doppler factor caused by a target's radial velocity relative to the observer. This doppler factor, D can be used to calculate the doppler shift caused by this velocity as $\lambda_r = \lambda_e D$ where λ_r is the wavelength received by the observer and λ_e is the wavelength emitted at the target.

This doppler factor is calculated as $D = \sqrt{\frac{1+v/c}{1-v/c}}$ where v is the input radial_velocity and c is the speed of light.

See also https://en.wikipedia.org/wiki/Relativistic_Doppler_effect#Relativistic_longitudinal_Doppler_effect

Parameters
radial_velocity – Radial velocity in km/s with positive values corresponding to motion away from the observer. This can be a single float value or a numpy array containing multiple velocity values.

Returns
Doppler factor calculated from input radial velocity. If the input radial_velocity is a single value, then a float is returned. If the input radial_velocity is a numpy array, then a numpy array of doppler factors is returned.

static load_spice_kernels(*kernel_path: str | None = None, manual_kernels: None | list[str] = None, only_if_needed: bool = True*) → None

Attempt to intelligently SPICE kernels using *planetmapper.base.load_kernels()*.

If manual_kernels is None (the default), then all kernels in the directory given by kernel_path which match the following patterns are loaded:

- `**/*.bsp`
- `**/*.tpc`
- `**/*.tls`

Note that these patterns match an arbitrary number of nested directories (within `kernel_path`). If more control is required, you can instead specify a list of specific kernels to load with `manual_kernels`.

Hint: See the *SPICE kernel documentation* for more detail about downloading SPICE kernels and the automatic kernel loading behaviour.

Parameters

- **`kernel_path`** – Path to directory where kernels are stored. If this is `None` (the default) then the result of *`get_kernel_path()`* is used. It is usually recommended to use one of the methods described in *the kernel directory documentation* rather than using this `kernel_path` argument.
- **`manual_kernels`** – Optional manual list of paths to kernels to load instead of using `kernel_path`.
- **`only_if_needed`** – If this is `True`, kernels will only be loaded once per session.

`static close_loop`(*arr: ndarray*) → ndarray

Return copy of array with first element appended to the end.

This is useful for cases like plotting the limb of a planet where the array of values forms a loop with the first and last values in `arr` adjacent to each other.

Parameters

`arr` – Array of values of length n.

Returns

Array of values of length $n + 1$ where the final value is the same as the first value.

`static unit_vector`(*v: ndarray*) → ndarray

Return normalised copy of a vector.

For an input vector $\vec{v}$, return the unit vector $\hat{v} = \frac{\vec{v}}{|\vec{v}|}$.

Parameters

`v` – Input vector to normalise.

Returns

Normalised vector which is parallel to `v` and has a magnitude of 1.

`static vector_magnitude`(*v: ndarray*) → float

Return the magnitude of a vector.

For an input vector $\vec{v}$, return magnitude $|\vec{v}| = \sqrt{\sum v_i^2}$.

Parameters

`v` – Input vector.

Returns

Magnitude (length) of vector.

static angular_dist(*ra1: float*, *dec1: float*, *ra2: float*, *dec2: float*) → float

Calculate the angular distance between two RA/Dec coordinate pairs.

Parameters

- **ra1** – RA of first point.
- **dec1** – Dec of first point.
- **ra2** – RA of second point
- **dec2** – Dec of second point.

Returns

Angular distance in degrees between the two points.

class planetmapper.**Body**(*target: str | int*, *utc: str | datetime.datetime | float | None = None*, *observer: str | int = 'EARTH'*, ***, *aberration_correction: str = 'CN'*, *observer_frame: str = 'J2000'*, *illumination_source: str = 'SUN'*, *subpoint_method: str = 'INTERCEPT/ELLIPSOID'*, *surface_method: str = 'ELLIPSOID'*, ***kwargs*)

Bases: *BodyBase*

Class representing an astronomical body observed at a specific time.

Generally only `target`, `utc` and `observer` need to be changed. The additional parameters allow customising the exact settings used in the internal SPICE functions. Similarly, some methods (e.g. *terminator_radec()*) have parameters that are passed to SPICE functions which can almost always be left as their default values. See the SPICE documentation for more details about possible parameter values.

The `target` and `observer` names are passed to *SpiceBase.standardise_body_name()*, so a variety of formats are acceptable. For example `'jupiter'`, `'JUPITER'`, `' Jupiter '`, `'599'` and `599` will all resolve to `'JUPITER'`.

Body instances are hashable, so can be used as dictionary keys.

This class inherits from *SpiceBase* so the methods described above are also available.

Parameters

- **target** – Name of target body.
- **utc** – Time of observation. This can be provided in a variety of formats and is assumed to be UTC unless otherwise specified. The accepted formats are: any `string` datetime representation compatible with SPICE (e.g. `'2000-12-31T23:59:59'` - see the documentation of acceptable string formats), a Python `datetime` object, or a `float` representing the Modified Julian Date (MJD) of the observation. Alternatively, if `utc` is `None` (the default), then the current time is used.
- **observer** – Name of observing body. Defaults to `'EARTH'`.
- **aberration_correction** – Aberration correction used to correct light travel time in SPICE. Defaults to `'CN'`.
- **observer_frame** – Observer reference frame. Defaults to `'J2000'`,
- **illumination_source** – Illumination source. Defaults to `'SUN'`.
- **subpoint_method** – Method used to calculate the sub-observer point in SPICE. Defaults to `'INTERCEPT/ELLIPSOID'`.
- **surface_method** – Method used to calculate surface intercepts in SPICE. Defaults to `'ELLIPSOID'`.
- ****kwargs** – Additional arguments are passed to *SpiceBase*.

target: *str*

Name of the target body, as standardised by *SpiceBase.standardise_body_name()*.

utc: *str*

String representation of the time of the observation in the format `'2000-01-01T00:00:00.000000'`. This time is in the UTC timezone.

observer: *str*

Name of the observer body, as standardised by *SpiceBase.standardise_body_name()*.

aberration_correction: *str*

Aberration correction used to correct light travel time in SPICE.

observer_frame: *str*

Observer reference frame.

et: *float*

Ephemeris time of the observation corresponding to `utc`.

dtm: *datetime.datetime*

Python timezone aware datetime of the observation corresponding to `utc`.

target_body_id: *int*

SPICE numeric ID of the target body.

target_light_time: *float*

Light time from the target to the observer at the time of the observation.

target_distance: *float*

Distance from the target to the observer at the time of the observation.

target_ra: *float*

Right ascension (RA) of the target centre.

target_dec: *float*

Declination (Dec) of the target centre.

illumination_source: *str*

Illumination source.

subpoint_method: *str*

Method used to calculate the sub-observer point in SPICE.

surface_method: *str*

Method used to calculate surface intercepts in SPICE.

radii: *np.ndarray*

Array of radii of the target body along the x, y and z axes in km.

r_eq: *float*

Equatorial radius of the target body in km.

r_polar: *float*

Polar radius of the target body in km.

flattening: *float*

Flattening of target body, calculated as `(r_eq - r_polar) / r_eq`.

prograde: bool

Boolean indicating if the target's spin sense is prograde or retrograde.

positive_longitude_direction: Literal['E', 'W']

Positive direction of planetographic longitudes. `'W'` implies positive west planetographic longitudes and `'E'` implies positive east longitudes.

This is determined from the target's spin sense (i.e. from *prograde*), with positive west longitudes for prograde rotation and positive east for retrograde. The earth, moon and sun are exceptions to this and are defined to have positive east longitudes

For more details, see https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/cspice/pgrrec_c.html#Particulars

target_diameter_arcsec: float

Equatorial angular diameter of the target in arcseconds.

This is calculated using `arcsin(body.r_eq / body.target_distance)`, so can underestimate the diameter if the observer is located very close to the target. If you require exact values for an observer close to the target, you can use the limb coordinates returned by *limb_radec()* to calculate the diameter.

subpoint_distance: float

Distance from the observer to the sub-observer point on the target.

subpoint_lon: float

Longitude of the sub-observer point on the target.

subpoint_lat: float

Latitude of the sub-observer point on the target.

subsol_lon: float

Longitude of the sub-solar point on the target.

subsol_lat: float

Latitude of the sub-solar point on the target.

named_ring_data: dict[str, list[float]]

Dictionary of ring radii for the target from *data_loader.get_ring_radii()*.

The dictionary keys are the names of the rings, and values are list of ring radii in km. If the length of this list is 2, then the values give the inner and outer radii of the ring respectively. Otherwise, the length should be 1, meaning the ring has a single radius. These ring radii values are sourced from planetary factsheets. If no ring data is available for the target, this dictionary is empty.

Values from this dictionary can be easily accessed using the convenience function *ring_radii_from_name()*.

ring_radii: set[float]

Set of ring radii in km to plot around the target body's equator. Each radius is plotted as a single line, so for a wide ring you may want to add both the inner and outer edger of the ring. The radii are defined as the distance from the centre of the target body to the ring. For Saturn, the A, B and C rings from *named_ring_data* are included by default. For all other bodies, `ring_radii` is empty by default.

Ring radii data from the *named_ring_data* can easily be added to `ring_radii` using *add_named_rings()*. Example usage:

```
body.ring_radii.add(122340) # Add new ring radius to plot
body.ring_radii.add(136780) # Add new ring radius to plot
body.ring_radii.update([66900, 74510]) # Add multiple radii to plot at once
```

(continues on next page)

(continued from previous page)

```
body.ring_radii.remove(122340) # Remove a ring radius
body.ring_radii.clear() # Remove all ring radii

# Add specific ring radii using data from planetary factsheets
body.add_named_rings('main', 'halo')

# Add all rings defined in the planetary factsheets
body.add_named_rings()
```

See also *ring_radec()*.

other_bodies_of_interest: list[*Body* | *BasicBody*]

List of other bodies of interest to mark when plotting. Add to this list using *add_other_bodies_of_interest()*.

coordinates_of_interest_lonlat: list[tuple[float, float]]

List of `(lon, lat)` coordinates of interest on the surface of the target body to mark when plotting (points which are not visible will not be plotted). To add a new point of interest, simply append a coordinate pair to this list:

```
body.coordinates_of_interest_lonlat.append((0, -22))
```

coordinates_of_interest_radec: list[tuple[float, float]]

List of `(ra, dec)` sky coordinates of interest to mark when plotting (e.g. background stars). To add new point of interest, simply append a coordinate pair to this list:

```
body.coordinates_of_interest_radec.append((200, -45))
```

create_other_body(*other_target: str | int*, *fallback_to_basic_body: Literal[False]*) → *Body*

create_other_body(*other_target: str | int*, *fallback_to_basic_body: bool = True*) → *planetmapper.body.Body | planetmapper.basic_body.BasicBody*

Create a *Body* instance using identical parameters but just with a different target. For example, the `europa` body created here will have identical parameters (see below) to the `jupiter` body, just with a different target.

```
jupiter = Body('Jupiter', '2000-01-01', observer='Moon')
europa = jupiter.create_other_body('Europa')
```

The parameters kept the same are `utc`, `observer`, `observer_frame`, `illumination_source`, `aberration_correction`, `subpoint_method`, and `surface_method`.

If a full *Body* instance cannot be created due to insufficient data in the SPICE kernel, a *BasicBody* instance will be created instead. This is useful for objects such as minor satellites which do not have known radius data.

Parameters

- **other_target** – Name of the other target, passed to *Body*
- **fallback_to_basic_body** – If a full *Body* instance cannot be created due to insufficient data in the SPICE kernel, attempt to create a *BasicBody* instance instead.

Returns

Body or *BasicBody* instance which corresponds to `other_target`.

add_other_bodies_of_interest(**other_targets: str | int*, *only_visible: bool = False*) → None

Add targets to the list of *other_bodies_of_interest* of interest to mark when plotting. The other targets are created using *create_other_body()*. For example, to add the Galilean moons as other targets to a Jupiter body, use

```
body = planetmapper.Body('Jupiter')
body.add_other_bodies_of_interest('Io', 'Europa', 'Ganymede', 'Callisto')
```

Integer SPICE ID codes can also be provided, which can be used to simplify adding multiple satellites to plots.

```
body = planetmapper.Body('Uranus')
body.add_other_bodies_of_interest(*range(701, 711))
# Uranus' satellites have ID codes 701, 702, 703 etc, so this adds 10 moons
# with a single function call
```

See also *add_satellites_to_bodies_of_interest()*.

Parameters

- ***other_targets** – Names of the other targets, passed to *Body*
- **only_visible** – If `True`, other targets which are hidden behind the target will not be added to *other_bodies_of_interest*.

add_satellites_to_bodies_of_interest(*skip_insufficient_data: bool = False*, *only_visible: bool = False*) → None

Automatically add all satellites in the target planetary system to *other_bodies_of_interest*.

This uses the NAIF ID codes to identify the satellites. For example, Uranus has an ID of 799, and its satellites have codes 701, 702, 703…, so any object with a code in the range 701 to 798 is added for Uranus.

See also *add_other_bodies_of_interest()*.

Parameters

- **skip_insufficient_data** – If True, satellites with insufficient data in the SPICE kernel will be skipped. If False, an exception will be raised.
- **only_visible** – If `True`, satellites which are hidden behind the target body will not be added.

ring_radii_from_name(*name: str*) → list[float]

Get list of ring radii in km for a named ring.

This is a convenience function to load data from *named_ring_data*.

Parameters

name – Name of ring. This is case insensitive and, "ring" suffix is optional and non-ASCII versions are allowed. For example, `'liberte'` will load the `'Liberté'` ring radii for Uranus and `'amalthea'` will load the `'Amalthea Ring'` radii for Jupiter.

Raises

`ValueError` – if no ring with the provided name is found.

Returns

List of ring radii in km. If the length of this list is 2, then the values give the inner and outer radii of the ring respectively. Otherwise, the length should be 1, meaning the ring has a single radius.

add_named_rings(**names: str*) → None

Add named rings to *ring_radii* so that they appear when creating wireframe plots. If no arguments are provided (i.e. calling `body.add_named_rings()`), then all rings in *named_ring_data* are added to *ring_radii*.

This is a convenience function to add data from *named_ring_data* to *ring_radii*.

Parameters

***names** – Ring names which are passed to *ring_radii_from_name()*. If no names are provided then all rings in *named_ring_data* are added.

lonlat2radec(*lon: float*, *lat: float*) → tuple[float, float]

Convert longitude/latitude coordinates on the target body to RA/Dec sky coordinates for the observer.

Parameters

- **lon** – Longitude of point on target body.
- **lat** – Latitude of point on target body.

Returns

`(ra, dec)` tuple containing the RA/Dec coordinates of the point.

radec2lonlat(*ra: float*, *dec: float*, *not_found_nan: bool = True*) → tuple[float, float]

Convert RA/Dec sky coordinates for the observer to longitude/latitude coordinates on the target body.

The provided RA/Dec will not necessarily correspond to any longitude/latitude coordinates, as they could be 'missing' the target and instead be observing the background sky. In this case, the returned longitude/latitude values will be NaN if `not_found_nan` is True (the default) or this function will raise an error if `not_found_nan` is False.

Parameters

- **ra** – Right ascension of point in the sky of the observer.
- **dec** – Declination of point in the sky of the observer.
- **not_found_nan** – Controls behaviour when the input `ra` and `dec` coordinates are missing the target body.

Returns

`(lon, lat)` tuple containing the longitude/latitude coordinates on the target body. If the provided RA/Dec coordinates are missing the target body and `not_found_nan` is True, then the `lon` and `lat` values will both be NaN.

Raises

NotFoundError – If the provided RA/Dec coordinates are missing the target body and `not_found_nan` is False, then NotFoundError will be raised.

lonlat2targvec(*lon: float*, *lat: float*) → ndarray

Convert longitude/latitude coordinates on the target body to rectangular vector centred in the target frame (e.g. for use as an input to a SPICE function).

Parameters

- **lon** – Longitude of point on target body.
- **lat** – Latitude of point on target body.

Returns

Numpy array corresponding to the 3D rectangular vector describing the longitude/latitude point in the target frame of reference.

targvec2lonlat(*targvec: ndarray*) → tuple[float, float]

Convert rectangular vector centred in the target frame to longitude/latitude coordinates on the target body (e.g. to convert the output from a SPICE function).

Parameters

targvec – 3D rectangular vector in the target frame of reference.

Returns

`(lon, lat)` tuple containing the longitude and latitude corresponding to the input vector.

radec2angular(*ra: float*, *dec: float*, *, *origin_ra: float | None = None*, *origin_dec: float | None = None*, *coordinate_rotation: float = 0.0*) → tuple[float, float]

Convert RA/Dec sky coordinates for the observer to relative angular coordinates.

The origin and rotation of the relative angular coordinates can be customised using the `origin_ra`, `origin_dec` and `coordinate_rotation` arguments. If these are not provided, the origin will be the centre of the target body and the rotation will be the same as in RA/Dec coordinates.

Parameters

- **ra** – Right ascension of point in the sky of the observer.
- **dec** – Declination of point in the sky of the observer.
- **origin_ra** – Right ascension (RA) of the origin of the relative angular coordinates. If `None`, the RA of the centre of the target body is used.
- **origin_dec** – Declination (Dec) of the origin of the relative angular coordinates. If `None`, the Dec of the centre of the target body is used.
- **coordinate_rotation** – Angle in degrees to rotate the relative angular coordinates around the origin, relative to the positive declination direction. The default `coordinate_rotation` is 0.0, so the target will have the same orientation as in RA/Dec coordinates.

Returns

`(angular_x, angular_y)` tuple containing the relative angular coordinates of the point in arcseconds.

angular2radec(*angular_x: float*, *angular_y: float*, ***angular_kwargs: Unpack[AngularCoordinateKwargs]*) → tuple[float, float]

Convert relative angular coordinates to RA/Dec sky coordinates for the observer.

Parameters

- **angular_x** – Angular coordinate in the x direction in arcseconds.
- **angular_y** – Angular coordinate in the y direction in arcseconds.
- ****angular_kwargs** – Additional arguments are used to customise the origin and rotation of the relative angular coordinates. See *radec2angular()* for details.

Returns

`(ra, dec)` tuple containing the RA/Dec coordinates of the point.

angular2lonlat(*angular_x: float*, *angular_y: float*, *, *not_found_nan: bool = True*, ***angular_kwargs: Unpack[AngularCoordinateKwargs]*) → tuple[float, float]

Convert relative angular coordinates to longitude/latitude coordinates on the target body.

Parameters

- **angular_x** – Angular coordinate in the x direction in arcseconds.

- **angular_y** – Angular coordinate in the y direction in arcseconds.
- **not_found_nan** – Controls behaviour when the input `angular_x` and `angular_y` coordinates are missing the target body.
- ****angular_kwargs** – Additional arguments are used to customise the origin and rotation of the relative angular coordinates. See *radec2angular()* for details.

Returns
`(lon, lat)` tuple containing the longitude and latitude of the point. If the provided angular coordinates are missing the target body and `not_found_nan` is True, then the `lon` and `lat` values will both be NaN.

Raises
NotFoundError – If the provided angular coordinates are missing the target body and `not_found_nan` is False, then NotFoundError will be raised.

lonlat2angular(*lon: float*, *lat: float*, ***angular_kwargs: Unpack[AngularCoordinateKwargs]*) → tuple[float, float]

Convert longitude/latitude coordinates on the target body to relative angular coordinates.

Parameters

- **lon** – Longitude of point on target body.
- **lat** – Latitude of point on target body.
- ****angular_kwargs** – Additional arguments are used to customise the origin and rotation of the relative angular coordinates. See *radec2angular()* for details.

Returns
`(angular_x, angular_y)` tuple containing the relative angular coordinates of the point in arcseconds.

km2radec(*km_x: float*, *km_y: float*) → tuple[float, float]

Convert distance in target plane to RA/Dec sky coordinates for the observer.

Parameters

- **km_x** – Distance in target plane in km in the East-West direction.
- **km_y** – Distance in target plane in km in the North-South direction.

Returns
`(ra, dec)` tuple containing the RA/Dec coordinates of the point.

radec2km(*ra: float*, *dec: float*) → tuple[float, float]

Convert RA/Dec sky coordinates for the observer to distances in the target plane.

Parameters

- **ra** – Right ascension of point in the sky of the observer.
- **dec** – Declination of point in the sky of the observer.

Returns
`(km_x, km_y)` tuple containing distances in km in the target plane in the East-West and North-South directions respectively.

km2lonlat(*km_x: float*, *km_y: float*, *not_found_nan: bool = True*) → tuple[float, float]

Convert distance in target plane to longitude/latitude coordinates on the target body.

Parameters

- **km_x** – Distance in target plane in km in the East-West direction.
- **km_y** – Distance in target plane in km in the North-South direction.
- **not_found_nan** – Controls behaviour when the input km_x and km_y coordinates are missing the target body.

Returns

(lon, lat) tuple containing the longitude and latitude of the point. If the provided km coordinates are missing the target body, then the lon and lat values will both be NaN if not_found_nan is True, otherwise a NotFoundError will be raised.

Raises

- **NotFoundError** – If the provided km coordinates are missing the target body
- **and not_found_nan is False, then NotFoundError will be raised.** –

lonlat2km(*lon: float*, *lat: float*) → tuple[float, float]

Convert longitude/latitude coordinates on the target body to distances in the target plane.

Parameters

- **lon** – Longitude of point on the target body.
- **lat** – Latitude of point on the target body.

Returns

(km_x, km_y) tuple containing distances in km in the target plane in the East-West and North-South directions respectively.

km2angular(*km_x: float*, *km_y: float*, ***angular_kwargs: Unpack[AngularCoordinateKwargs]*) → tuple[float, float]

Convert distance in target plane to relative angular coordinates.

Parameters

- **km_x** – Distance in target plane in km in the East-West direction.
- **km_y** – Distance in target plane in km in the North-South direction.
- ****angular_kwargs** – Additional arguments are used to customise the origin and rotation of the relative angular coordinates. See *radec2angular()* for details.

Returns

(angular_x, angular_y) tuple containing the relative angular coordinates of the point in arcseconds.

angular2km(*angular_x: float*, *angular_y: float*, ***angular_kwargs: Unpack[AngularCoordinateKwargs]*) → tuple[float, float]

Convert relative angular coordinates to distances in the target plane.

Parameters

- **angular_x** – Angular coordinate in the x direction in arcseconds.
- **angular_y** – Angular coordinate in the y direction in arcseconds.
- ****angular_kwargs** – Additional arguments are used to customise the origin and rotation of the relative angular coordinates. See *radec2angular()* for details.

Returns

(km_x, km_y) tuple containing distances in km in the target plane in the East-West and North-South directions respectively.

limb_radec(****kwargs*) → tuple[numpy.ndarray, numpy.ndarray]

Calculate the RA/Dec coordinates of the target body's limb.

Parameters
npts – Number of points in the generated limb.

Returns
`(ra, dec)` tuple of coordinate arrays.

limb_radec_by_illumination(****kwargs*) → tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray]

Calculate RA/Dec coordinates of the dayside and nightside parts of the target body's limb.

Output arrays are like the outputs of *limb_radec()*, but the dayside coordinate arrays have non-illuminated locations replaced with NaN and the nightside arrays have illuminated locations replaced with NaN.

Parameters
npts – Number of points in the generated limbs.

Returns
`(ra_day, dec_day, ra_night, dec_night)` tuple of coordinate arrays of the dayside then nightside parts of the limb.

limb_coordinates_from_radec(*ra: float*, *dec: float*) → tuple[float, float, float]

Calculate the coordinates relative to the target body's limb for a point in the sky.

The coordinates are calculated for the point on the ray (as defined by RA/Dec) which is closest to the target body's limb.

Parameters

- **ra** – Right ascension of point in the sky of the observer.
- **dec** – Declination of point in the sky of the observer.

Returns
`(lon, lat, dist)` tuple of coordinates relative to the target body's limb. `lon` and `lat` give the planetographic longitude and latitude of the point on the limb closest to the point defined by `ra` and `dec`. `dist` gives the distance from the point defined by `ra` and `dec` to the target's limb. Positive values of `dist` mean that the point is above the limb and negative values mean that the point is below the limb (i.e. on the target body's disc).

test_if_lonlat_visible(*lon: float*, *lat: float*) → bool

Test if longitude/latitude coordinate on the target body are visible.

Parameters

- **lon** – Longitude of point on target body.
- **lat** – Latitude of point on target body.

Returns
True if the point is visible from the observer, otherwise False.

other_body_los_intercept(*other: str | int | planetmapper.body.Body | planetmapper.basic_body.BasicBody*) → Union[None, Literal['transit', 'hidden', 'part transit', 'part hidden', 'same']]

Test for line-of-sight intercept between the target body and another body.

This can be used to test for if another body (e.g. a moon) is in front of or behind the target body (e.g. a planet).

See also *test_if_other_body_visible()*.

Warning: This method does not perform any checks to ensure that any input *Body* or *BasicBody* instances have a consistent observer location and observation time as the target body.

Parameters

other – Other body to test for intercept with. Can be a :class`Body` (or *BasicBody*) instance, or a string/integer NAIF ID code which is passed to *create_other_body()*.

Returns

`None` if there is no intercept, otherwise a string indicating the type of intercept. For example, with `jupiter.other_body_los_intercept('europa')`, the possible return values mean:

- `None` - there is no intercept, meaning that Europa and Jupiter do not overlap in the sky.
- `'hidden'` - all of Europa's disk is hidden behind Jupiter.
- `'part hidden'` - part of Europa's disk is hidden behind Jupiter and part is visible.
- `'transit'` - all of Europa's disk is in front of Jupiter.
- `'part transit'` - part of Europa's disk is in front of Jupiter.

The return value can also be `'same'`, which means that the other body is the same object as the target body (or has an identical location).

test_if_other_body_visible(*other: str | int | planetmapper.body.Body | planetmapper.basic_body.BasicBody*) → bool

Test if another body is visible, or is hidden behind the target body.

This is a convenience method equivalent to:

```
body.other_body_los_intercept(other) != 'hidden'
```

Parameters

other – Other body to test for visibility, passed to *other_body_los_intercept()*.

Returns

`False` if the other body is hidden behind the target body, otherwise `True`. If any part of the other body is visible, this method will return `True`.

illumination_angles_from_lonlat(*lon: float*, *lat: float*) → tuple[float, float, float]

Calculate the illumination angles of a longitude/latitude coordinate on the target body.

Parameters

- **lon** – Longitude of point on target body.
- **lat** – Latitude of point on target body.

Returns

`(phase, incidence, emission)` tuple containing the illumination angles.

azimuth_angle_from_lonlat(*lon: float*, *lat: float*) → float

Calculate the azimuth angle of a longitude/latitude coordinate on the target body.

Parameters

- **lon** – Longitude of point on target body.

- **lat** – Latitude of point on target body.

Returns
Azimuth angle in degrees.

local_solar_time_from_lon(*lon: float*) → float

Calculate the numerical local solar time for a longitude on the target body. For example, `0.0` corresponds to midnight and `12.5` corresponds to 12:30pm.

See also *local_solar_time_string_from_lon()*.

Note: A 'local hour' of solar time is a defined as 1/24th of the solar day on the target body, so will not correspond to a 'normal' hour as measured by a clock. See the SPICE documentation for more details.

Parameters
lon – Longitude of point on target body.

Returns
Numerical local solar time in 'local hours'.

local_solar_time_string_from_lon(*lon: float*) → str

Local solar time string representation for a longitude on the target body. For example, `'00:00:00'` corresponds to midnight and `'12:30:00'` corresponds to 12:30pm.

See *local_solar_time_from_lon()* for more details.

Parameters
lon – Longitude of point on target body.

Returns
String representation of local solar time.

terminator_radec(*npts: int = 360*, *only_visible: bool = True*, *close_loop: bool = True*, *method: str = 'UMBRAL/TANGENT/ELLIPSOID'*, *corloc: str = 'ELLIPSOID TERMINATOR'*) → tuple[numpy.ndarray, numpy.ndarray]

Calculate the RA/Dec coordinates of the terminator (line between day and night) on the target body. By default, only the visible part of the terminator is returned (this can be changed with `only_visible`).

Parameters

- **npts** – Number of points in generated terminator.
- **only_visible** – Toggle only returning visible part of terminator.
- **close_loop** – If True, passes coordinate arrays through `close_loop()` (e.g. to enable nicer plotting).
- **method** – Passed to SPICE function.
- **corloc** – Passed to SPICE function.

Returns
`(ra, dec)` tuple of RA/Dec coordinate arrays.

test_if_lonlat_illuminated(*lon: float*, *lat: float*) → bool

Test if longitude/latitude coordinate on the target body are illuminated.

Parameters

- **lon** – Longitude of point on target body.

- **lat** – Latitude of point on target body.

Returns
True if the point is illuminated, otherwise False.

ring_plane_coordinates(*ra: float*, *dec: float*, *only_visible: bool = True*) → tuple[float, float, float]

Calculate coordinates in the target body's equatorial (ring) plane. This is mainly useful for calculating the coordinates in a body's ring system at a given point in the sky.

To calculate the coordinates corresponding to a location on the target body, you can use

```
body.ring_plane_coordinates(*body.radec2lonlat(lon, lat))
```

This form can be useful to identify parts of a planet's surface which are obscured by its rings

```
radius, _, _ = body.ring_plane_coordinates(*body.lonlat2radec(lon, lat))
ring_data = planetmapper.data_loader.get_ring_radii()['SATURN']
for name, radii in ring_data.items():
    if min(radii) < radius < max(radii):
        print(f'Point obscured by {name} ring')
        break
else:
    print('Point not obscured by rings')
```

Parameters

- **ra** – Right ascension of point in the sky of the observer.
- **dec** – Declination of point in the sky of the observer.
- **only_visible** – If `True` (the default), coordinates for parts of the equatorial plane hidden behind the target body are set to NaN.

Returns
`(ring_radius, ring_longitude, ring_distance)` tuple for the point on the target body's equatorial (ring) plane. `ring_radius` gives the distance of the point in km from the centre of the target body. `ring_longitude` gives the planetographic longitude of the point in degrees. `ring_distance` gives the distance from the observer to the point in km.

ring_radec(*radius: float*, *npts: int = 360*, *only_visible: bool = True*) → tuple[numpy.ndarray, numpy.ndarray]

Calculate RA/Dec coordinates of a ring around the target body.

The ring is assumed to be directly above the planet's equator and has a constant `radius` for all longitudes. Use *ring_radii* to set the rings automatically plotted.

Parameters

- **radius** – Radius in km of the ring from the centre of the target body.
- **npts** – Number of points in the generated ring.
- **only_visible** – If `True` (default), the coordinates for the part of the ring hidden behind the target body are set to NaN. This routine will execute slightly faster with `only_visible` set to `False`.

Returns
`(ra, dec)` tuple of coordinate arrays.

visible_lonlat_grid_radec(*interval: float = 30*, ***kwargs*) → list[tuple[numpy.ndarray, numpy.ndarray]]

Convenience function to calculate a grid of equally spaced lines of constant longitude and latitude for use in plotting lon/lat grids.

This function effectively combines *visible_lon_grid_radec()* and *visible_lat_grid_radec()* to produce both longitude and latitude gridlines.

For example, to plot gridlines with a 45 degree interval, use:

```
lines = body.visible_lonlat_grid_radec(interval=45)
for ra, dec in lines:
    plt.plot(ra, dec)
```

Parameters

- **interval** – Spacing of gridlines. Generally, this should be an integer factor of 90 to produce nice looking plots (e.g. 10, 30, 45 etc).
- ****kwargs** – Additional arguments are passed to *visible_lon_grid_radec()* and *visible_lat_grid_radec()*.

Returns

List of `(ra, dec)` tuples, each of which corresponds to a gridline. `ra` and `dec` are arrays of RA/Dec coordinate values for that gridline.

visible_lon_grid_radec(*lons: list[float] | numpy.ndarray*, *npts: int = 60*, ***, *lat_limit: float = 90*) → list[tuple[numpy.ndarray, numpy.ndarray]]

Calculates the RA/Dec coordinates for visible lines of constant longitude.

For each longitude in `lons`, a `(ra, dec)` tuple is calculated which contains arrays of RA and Dec coordinates. Coordinates which correspond to points which are not visible are replaced with NaN.

See also *visible_lonlat_grid_radec()*,

Parameters

- **lons** – List of longitudes to plot.
- **npts** – Number of points in each full line of constant longitude.
- **lat_limit** – Latitude limit for gridlines. For example, if `lat_limit=60`, the gridlines will be calculated for latitudes between 60°N and 60°S (inclusive).

Returns

List of `(ra, dec)` tuples, corresponding to the list of input `lons`. `ra` and `dec` are arrays of RA/Dec coordinate values for that gridline.

visible_lat_grid_radec(*lats: list[float] | numpy.ndarray*, *npts: int = 120*, ***, *lat_limit: float = 90*) → list[tuple[numpy.ndarray, numpy.ndarray]]

Constant latitude version of *visible_lon_grid_radec()*. See also *visible_lonlat_grid_radec()*.

Parameters

- **lats** – List of latitudes to plot.
- **npts** – Number of points in each full line of constant latitude.
- **lat_limit** – Latitude limit for gridlines. For example, if `lat_limit=60`, only gridlines with latitudes between 60°N and 60°S (inclusive) will be calculated.

Returns
List of `(ra, dec)` tuples, corresponding to the list of input `lats`. `ra` and `dec` are arrays of RA/Dec coordinate values for that gridline.

`radial_velocity_from_lonlat`(*lon: float*, *lat: float*) → float

Calculate radial (i.e. line-of-sight) velocity of a point on the target's surface relative to the observer. This can be used to calculate the doppler shift.

Parameters

- **lon** – Longitude of point on target body.
- **lat** – Latitude of point on target body.

Returns
Radial velocity of the point in km/s.

`distance_from_lonlat`(*lon: float*, *lat: float*) → float

Calculate distance from observer to a point on the target's surface.

Parameters

- **lon** – Longitude of point on target body.
- **lat** – Latitude of point on target body.

Returns
Distance of the point in km.

`graphic2centric_lonlat`(*lon: float*, *lat: float*) → tuple[float, float]

Convert planetographic longitude/latitude to planetocentric.

Parameters

- **lon** – Planetographic longitude.
- **lat** – Planetographic latitude.

Returns
`(lon_centric, lat_centric)` tuple of planetocentric coordinates.

`centric2graphic_lonlat`(*lon_centric: float*, *lat_centric: float*) → tuple[float, float]

Convert planetocentric longitude/latitude to planetographic.

Parameters

- **lon_centric** – Planetocentric longitude.
- **lat_centric** – Planetographic latitude.

Returns
`(lon, lat)` tuple of planetographic coordinates.

`north_pole_angle`() → float

Calculate the angle of the north pole of the target body relative to the positive declination direction.

Note: This method calculates the angle between the centre of the target and its north pole, so may produce unexpected results for targets which are located directly at the celestial pole.

Returns
Angle of the north pole in degrees (-180 to 180).

get_description(*multiline: bool = True*) → str

Generate a useful description of the body.

Parameters
multiline – Toggles between multi-line and single-line version of the description.

Returns
String describing the observation of the body.

get_poles_to_plot() → list[tuple[float, float, str]]

Get list of poles on the target body for use in plotting.

If at least one pole is visible, return the visible poles. If no poles are visible, return both poles but in brackets. This ensures that at lease one pole is always returned (to orientate the observation).

Returns
List of `(lon, lat, label)` tuples describing the poles where `lon` and `lat` give the coordinates of the pole on the target and `label` is a string describing the pole. If the pole is visible, the `label` is either 'N' or 'S'. If neither pole is visible, then both poles are returned with labels of '(N)' and '(S)'.

matplotlib_radec2km_transform(*ax: matplotlib.axes._axes.Axes | None = None*) → Transform

Get matplotlib transform which converts between coordinate systems.

For example, *matplotlib_radec2km_transform()* can be used to plot data in RA/Dec coordinates directly on a plot in the km coordinate system:

```
# Create plot in km coordinates
ax = body.plot_wireframe_km()

# Plot data using RA/Dec coordinates with the transform
ax.scatter(
    body.target_ra,
    body.target_dec,
    transform=body.matplotlib_radec2km_transform(ax),
    color='r',
)
# This is (almost exactly) equivalent to using
# ax.scatter(*body.radec2km(body.target_ra, body.target_dec), color='r')
```

A full set of transformations are available in *Body* (below) and *BodyXY* to convert between various coordinate systems. These are mainly convenience functions to simplify plotting data in different coordinate systems, and may not be exact in some extreme geometries, due to the non-linear nature of spherical coordinates.

Warning: The transformations are performed as affine transformations, which are linear transformations. This means that the transformations may be inexact at large distances from the target body, or near the celestial poles for `radec` coordinates.

For the vast majority of purposes, these matplotlib transformations are accurate, but if you are working with extreme geometries or require exact transformations you should convert the coordinates manually before plotting (e.g. using *radec2km()* rather than *matplotlib_radec2km_transform()*).

The `km`, `angular` (with the default values for the origin) and `xy` coordinate systems are all affine transformations of each other, so the matplotlib transformations between these coordinate systems should be exact.

Parameters

ax – Optionally specify a matplotlib axis to return `transform_radec2km + ax.transData`. This value can then be used in the `transform` keyword argument of a Matplotlib function without any further modification.

Returns

Matplotlib transformation from `radec` to `km` coordinates.

matplotlib_km2radec_transform(*ax: matplotlib.axes._axes.Axes | None = None*) → Transform

matplotlib_radec2angular_transform(*ax: matplotlib.axes._axes.Axes | None = None*, ***angular_kwargs: Unpack[AngularCoordinateKwargs]*) → Transform

matplotlib_angular2radec_transform(*ax: matplotlib.axes._axes.Axes | None = None*, ***angular_kwargs: Unpack[AngularCoordinateKwargs]*) → Transform

plot_wireframe_radec(*ax: matplotlib.axes._axes.Axes | None = None*, ***, *scale_factor: float | None = None*, *dms_ticks: bool | None = None*, *add_axis_labels: bool | None = None*, *aspect_adjustable: Optional[Literal['box', 'datalim']] = 'datalim'*, *use_shifted_meridian: bool = False*, *show: bool = False*, ***wireframe_kwargs: Unpack[WireframeKwargs]*) → Axes

Plot basic wireframe representation of the observation using RA/Dec sky coordinates.

See also *plot_wireframe_km()*, *plot_wireframe_angular()* and *BodyXY.plot_wireframe_xy()* to plot the wireframe in other coordinate systems. *plot_wireframe_custom()* can also be used to plot a wireframe with a custom coordinate system.

Hint: See *the examples page* for more examples of creating wireframe plots.

To plot a wireframe with the default appearance, simply use:

```
body.plot_wireframe_radec()
```

To customise the appearance of the plot, you can use the `formatting` and `**kwargs` arguments which can be used to pass arguments to the Matplotlib plotting functions. The `formatting` argument can be used to customise individual components, and the `**kwargs` argument can be used to customise all components at once.

For example, to change the colour of the entire wireframe to red, you can use:

```
body.plot_wireframe_radec(color='r')
```

To change just the plotted terminator and dayside limb to red, use:

```
body.plot_wireframe_radec(
    formatting={
        'terminator': {'color': 'r'}, 'limb_illuminated': {'color': 'r'},
    },
)
```

The order of precedence for the formatting is the `formatting` argument, then `**kwargs`, then the default formatting. For example, the following plot will be red with a thin blue grid and green poles:

```
body.plot_wireframe_radec(
    color='r', formatting={
        'grid': {'color': 'b', 'linewidth': 0.5, 'linestyle': '-'}, 'pole':
        {'color': 'g'},
    },
)
```

Individual components can be hidden by setting `visible` to `False`. For example, to hide the terminator, use:

```
body.plot_wireframe_radec(formatting={'terminator': {'visible': False}})
```

The default formatting is defined in `DEFAULT_WIREFRAME_FORMATTING`. This can be modified after importing PlanetMapper to change the default appearance of all wireframes:

```
import planetmapper
planetmapper.DEFAULT_WIREFRAME_FORMATTING['grid']['color'] = 'b'
planetmapper.DEFAULT_WIREFRAME_FORMATTING['grid']['linestyle'] = '--'

body.plot_wireframe_radec() # This would have a blue dashed grid
body.plot_wireframe_radec(color='r') # This would be red with a dashed grid
```

The units of the plotted data can be customised with the `scale_factor` argument, which multiplies coordinates by the given `scale_factor` before plotting. For example:

```
body.plot_wireframe_radec() # units of degrees
body.plot_wireframe_radec(scale_factor=3.14159/180) # units of radians

body.plot_wireframe_km() # units of km
body.plot_wireframe_km(scale_factor=1000) # units of m
body.plot_wireframe_km(scale_factor=1/body.r_eq) # units of planet radii

body.plot_wireframe_angular() # units of arcseconds
body.plot_wireframe_angular(scale_factor=1/60) # units of arcminutes
body.plot_wireframe_angular(scale_factor=1/3600) # units of degrees
```

Warning: Even though the numerical values will be correct, the plot may appear warped or distorted if the target is near the celestial pole (i.e. the target's declination is near 90° or -90°). This is due to the spherical nature of the RA/Dec coordinate system, which is impossible to represent perfectly on a 2D cartesian plot.

plot_wireframe_angular() can be used as an alternative to *plot_wireframe_radec()* to plot the wireframe without distortion from the choice of coordinate system. By default, the `angular` coordinate system is centred on the target body, which minimises any distortion, but the origin and rotation of the `angular` coordinates can also be customised as needed (e.g. to align it with an instrument's field of view).

Note: If the target body is near RA=0°, then the wireframe may be split over two halves of the plot. This can be fixed by using `body.plot_wireframe_radec(use_shifted_meridian=True)`, which will plot the wireframe with RA coordinates between -180° and 180°, rather than the default of 0° to 360°.

Parameters

- **ax** – Matplotlib axis to use for plotting. If `ax` is None (the default), uses `plt.gca()` to get the currently active axis.
- **scale_factor** – Custom scale factor to apply to the plotted wireframe. This can be used to change units of the plot. If `scale_factor` is used, the plotted coordinates will be multiplied by `scale_factor` before plotting. See the examples above for more details.
- **label_poles** – Toggle labelling the poles of the target body.
- **add_title** – Add title generated by *get_description()* to the axis.
- **add_axis_labels** – Add axis labels to the plot. If `add_axis_labels` is None (the default), then labels will only be added if `scale_factor` is not used.
- **grid_interval** – Spacing between grid lines in degrees.
- **grid_lat_limit** – Latitude limit for gridlines. For example, if `grid_lat_limit=60`, then gridlines will only be plotted for latitudes between 60°N and 60°S (inclusive). This can be useful to reduce visual clutter around the poles.
- **indicate_equator** – Toggle indicating the equator with a solid line.
- **indicate_prime_meridian** – Toggle indicating the prime meridian with a solid line.
- **aspect_adjustable** – Set `adjustable` parameter when setting the aspect ratio. Passed to `matplotlib.axes.Axes.set_aspect()`. Set to None to skip setting the aspect ratio (generally this is only recommended if you're setting the aspect ratio yourself).
- **dms_ticks** – Toggle between showing ticks as degrees, minutes and seconds (e.g. 12°34′56″) or decimal degrees (e.g. 12.582). This argument is only applicable for *plot_wireframe_radec()*. If `dms_ticks` is None (the default), then ticks will only be shown as degrees, minutes and seconds if `scale_factor` is not used.
- **use_shifted_meridian** – If `use_shifted_meridian=True`, plot the wireframe with RA coordinates between -180° and 180°, rather than the default of 0° to 360°. This can be useful for bodies which lie at RA=0°, which can be split over two halves of the plot with the default `use_shifted_meridian=False`. This argument is only applicable for *plot_wireframe_radec()*.
- **show** – Toggle immediately showing the plotted figure with `plt.show()`.
- **formatting** – Dictionary of formatting options for the wireframe components. The keys of this dictionary are the names of the wireframe components and the values are dictionaries of keyword arguments to pass to the Matplotlib plotting function for that component. For example, to set the `color` of the plotted rings to red, you could use:

```
body.plot_wireframe_radec(formatting={'ring': {'color': 'r'}})
```

  The following components can be formatted: `grid`, `equator`, `prime_meridian`, `limb`, `limb_illuminated`, `terminator`, `ring`, `pole`, `coordinate_of_interest_lonlat`, `coordinate_of_interest_radec`, `other_body_of_interest_marker`, `other_body_of_interest_label`, `hidden_other_body_of_interest_marker`, `hidden_other_body_of_interest_label`.
- ****kwargs** – Additional arguments are passed to Matplotlib plotting functions for all components. This is useful for specifying properties like `color` to customise the entire wireframe rather than a single component. For example, to make the entire wireframe red, you could use:

```
body.plot_wireframe_radec(color='r')
```

Returns
The axis containing the plotted wireframe.

plot_wireframe_km(*ax: matplotlib.axes._axes.Axes | None = None*, ***, *scale_factor: float | None = None*, *add_axis_labels: bool | None = None*, *aspect_adjustable: Optional[Literal['box', 'datalim']] = 'datalim'*, *show: bool = False*, ***wireframe_kwargs: Unpack[WireframeKwargs]*) → Axes

Plot basic wireframe representation of the observation on a target centred frame. See *plot_wireframe_radec()* for details of accepted arguments.

Returns
The axis containing the plotted wireframe.

plot_wireframe_angular(*ax: matplotlib.axes._axes.Axes | None = None*, ***, *origin_ra: float | None = None*, *origin_dec: float | None = None*, *coordinate_rotation: float = 0.0*, *scale_factor: float | None = None*, *add_axis_labels: bool | None = None*, *aspect_adjustable: Optional[Literal['box', 'datalim']] = 'datalim'*, *show: bool = False*, ***wireframe_kwargs: Unpack[WireframeKwargs]*) → Axes

Plot basic wireframe representation of the observation on a relative angular coordinate frame. See *plot_wireframe_radec()* for details of accepted arguments.

The `origin_ra`, `origin_dec` and `coordinate_rotation` arguments can be used to customise the origin and rotation of the relative angular coordinate frame (see see *radec2angular()*). For example, to plot the wireframe with the origin at the north pole, you can use:

```
body.plot_wireframe_angular(origin_ra=0, origin_dec=90)
```

Warning: If custom values for `origin_ra` and `origin_dec` are provided, the plot may appear warped or distorted if the target is a large distance from the origin. This is because spherical coordinates are impossible to represent perfectly on a 2D cartesian plot. By default, the `angular` coordinates are centred on the target body, minimising any distortion.

Returns
The axis containing the plotted wireframe.

plot_wireframe_custom(*ax: matplotlib.axes._axes.Axes | None = None*, *coordinate_func: Optional[Callable[[float, float], tuple[float, float]]] = None*, ***, *transform: matplotlib.transforms.Transform | None = None*, *additional_array_func: Optional[Callable[[Iterable, Iterable], tuple[numpy.ndarray, numpy.ndarray]]] = None*, ***wireframe_kwargs: Unpack[WireframeKwargs]*) → Axes

Plot a custom wireframe representation of the observation, using a user-defined coordinate system.

This can be used to create a custom wireframe plot variant, similar to the *plot_wireframe_radec()*, *plot_wireframe_km()*, *plot_wireframe_angular()* and *BodyXY.plot_wireframe_xy()* methods. All wireframe variants use the same plotting code internally, and this method allows the internal wireframe plotting code to be accessed directly, with custom arguments. Most wireframe uses are covered by the built-in wireframe plotting methods but this method can be useful when plotting with custom projections or complex coordinate systems.

Hint: If you just want to change the units of a wireframe plot, this can be done with

the `scale_factor` argument of the built-in wireframe plotting methods. For example, `body.plot_wireframe_angular(scale_factor=1/60)` will plot the wireframe with units of arcminutes (rather than the default arcseconds).

The `coordinate_func` and `transform` arguments are used to convert data in RA/Dec coordinates into the desired coordinate system and apply any additional Matplotlib transforms desired to the plotted data. Both of these arguments are optional, so generally you will only need to specify a value for `coordinate_func`.

For example, this approximately replicates the *plot_wireframe_km()* method, by using *radec2km()* to convert RA/Dec coordinates to km coordinates:

```
ax = body.plot_wireframe_custom(coordinate_func=body.radec2km)
ax.set_aspect(1)
ax.set_xlabel('Projected distance (km)')
ax.set_ylabel('Projected distance (km)')
```

Or to plot a wireframe in custom 'angular' coordinates that are reflected in the y direction, you could use:

```
def coordinate_func(ra, dec):
    x, y = body.radec2angular(ra, dec)
    return x, -y

ax = body.plot_wireframe_custom(coordinate_func=coordinate_func)
ax.set_aspect(1)
```

The `transform` argument is mainly useful if you wish to create an interactive wireframe plot, where the plotted data can be changed after plotting (like in the PlanetMapper GUI). If both `coordinate_func` and `transform` are provided, then the `transform` is applied to the plotted data after transforming with `coordinate_func`. The plotting functionality when both `coordinate_func` and `transform` are provided can therefore be simplified as:

```
x, y = coordinate_func(ra, dec)
ax.scatter(x, y, transform=transform)
```

The `additional_array_func` argument can be used to specify a function to apply to arrays before plotting any linear features (e.g. the limb, gridlines, rings). For example, this is used internally in *plot_wireframe_radec()* to add NaNs into arrays of data whenever the coordinates wrap from one side of the axis to the other (to prevent lines being drawn across the entire axis). If specified, this function is applied after first converting the data with `coordinate_func` and before applying any `transform` argument, and is only applied to data plotted with Matplotlib's `plot` function. The plotting functionality when `coordinate_func`, `transform` and `additional_array_func` are provided can therefore be simplified as:

```
# plotting arrays of ra and dec coordinates
xs, ys = zip(*(coordinate_func(ra, dec) for ra, dec in zip(ras, decs)))
xs, ys = additional_array_func(xs, ys)
ax.plot(xs, ys, transform=transform)

# plotting individual ra and dec coordinates
x, y = coordinate_func(ra, dec)
ax.scatter(x, y, transform=transform)
```

Note: This method does not set the aspect ratio of the plot, so you will usually need to do this yourself to

ensure the plot is not distorted. For example, to set the aspect ratio to 1, you can use `ax.set_aspect(1)`.

Parameters

- **ax** – Matplotlib axis to use for plotting. If `ax` is None (the default), uses `plt.gca()` to get the currently active axis.
- **coordinate_func** – Function to convert RA/Dec coordinates to the desired coordinate system. Takes two arguments (RA, Dec) and returns two values (x, y). If this is not provided, then the default no-op function `coordinate_func=lambda ra, dec: (ra, dec)` is used.
- **transform** – Matplotlib transform to apply to the plotted data, after transforming with `coordinate_func`. If this is not provided, then no additional transform is applied.
- **additional_array_func** – Optional function to apply to iterable of converted (x, y) coordinates before plotting any linear features (e.g. the limb, gridlines, rings). This should take two iterables of x and y coordinates and return two arrays x and y coordinates to plot. The lengths of the input coordinates do not have to be the same as the lengths of the output coordinates, so `additional_array_func` can be used to add or remove points from the plotted data as needed. However, the length of the output x array should be the same as the length of the output y array. If this is not provided, then no additional function is applied.
- ****wireframe_kwargs** – See *plot_wireframe_radec()* for details of additional arguments.

class planetmapper.**Backplane**(*name: str*, *description: str*, *get_img: Callable[[], ndarray]*, *get_map: _BackplaneMapGetter*)

Bases: `NamedTuple`

NamedTuple containing information about a backplane.

Backplanes provide a way to generate and save additional information about an observation, such as the longitudes/latitudes corresponding to each pixel in the observed image. This class provides a standardised way to store a backplane generation function, along with some metadata (`name` and `description`) which describes what the backplane represents.

See also *BodyXY.backplanes*.

Parameters

- **name** – Short name identifying the backplane. This is used as the `EXTNAME` for the backplane when saving FITS files in `Observation.save()`.
- **description** – More detailed description of the backplane (e.g. including units).
- **get_img** – Function which takes no arguments returns a numpy array containing a backplane image when called. This should generally be a method such as *BodyXY.get_lon_img()*.
- **get_map** – Function returns a numpy array containing a map of backplane values when called. This should take map projection keyword arguments, as described in *BodyXY.generate_map_coordinates()*. This function should generally be a method such as *BodyXY.get_lon_map()*.

name: `str`

Alias for field number 0

description: `str`

Alias for field number 1

get_img: `Callable[[], ndarray]`

Alias for field number 2

get_map: `_BackplaneMapGetter`

Alias for field number 3

class planetmapper.**BodyXY**(*target: str*, *utc: str | datetime.datetime | float | None = None*, *nx: int = 0*, *ny: int = 0*, ***, *sz: int | None = None*, ***kwargs*)

Bases: *Body*

Class representing an astronomical body imaged at a specific time.

This is a subclass of *Body* with additional methods to interact with the image pixel coordinate frame `xy`. This class assumes the tangent plane approximation is valid for the conversion between pixel coordinates `xy` and RA/Dec sky coordinates `radec`.

The `xy` `radec` conversion is performed by setting the pixel coordinates of the centre of the planet's disc `(x0, y0)`, the (equatorial) pixel radius of the disc `r0` and the `rotation` of the disc. These disc parameters can be adjusted using methods such as *set_x0()* and retrieved using methods such as *get_x0()*. It is important to note that conversions involving `xy` image pixel coordinates (e.g. backplane image generation) will produce different results before and after these disc parameter values are adjusted.

For larger images, the generation of backplane images can be computationally intensive and take a large amount of time to execute. Therefore, intermediate results are cached to make sure that the slowest parts of code are only called when needed. This cache is managed automatically, so the user never needs to worry about dealing with it. The cache behaviour can be seen in apparently similar lines of code having very different execution times:

```
# Create a new object
body = planetmapper.BodyXY('Jupiter', '2000-01-01', sz=500)
body.set_disc_params(x0=250, y0=250, r0=200)
# At this point, the cache is completely empty

# The intermediate results used in generating the incidence angle backplane
# are cached, speeding up any future calculations which use these
# intermediate results:
body.get_backplane_img('INCIDENCE') # Takes ~10s to execute
body.get_backplane_img('INCIDENCE') # Executes instantly
body.get_backplane_img('EMISSION') # Executes instantly

# When any of the disc parameters are changed, the xy <-> radec conversion
# changes so the cache is automatically cleared (as the cached intermediate
# results are no longer valid):
body.set_r0(190) # This automatically clears the cache
body.get_backplane_img('EMISSION') # Takes ~10s to execute
body.get_backplane_img('INCIDENCE') # Executes instantly
```

You can optionally display a progress bar for long running processes like backplane generation by `show_progress=True` when creating a `BodyXY` instance (or any other instance which derives from *SpiceBase*).

The size of the image can be specified by using the `nx` and `ny` parameters to specify the number of pixels in the x and y dimensions of the image respectively. If `nx` and `ny` are equal (i.e. the image is square), then the parameter `sz` can be used instead to set both `nx` and `ny`, where `BodyXY(..., sz=50)` is equivalent to `BodyXY(..., nx=50, ny=50)`.

If `nx` and `ny` are not set, then some functionality (such as generating backplane images) will not be available and will raise a `ValueError` if called.

BodyXY instances are mutable and therefore not hashable, meaning that they cannot be used as dictionary keys. *to_body()* can be used to create a *Body* instance which is hashable.

Parameters

- **target** – Name of target body, passed to *Body*.
- **utc** – Time of observation, passed to *Body*.
- **observer** – Name of observing body, passed to *Body*.
- **nx** – Number of pixels in the x dimension of the image.
- **ny** – Number of pixels in the y dimension of the image.
- **sz** – Convenience parameter to set both `nx` and `ny` to the same value. `BodyXY(..., sz=50)` is equivalent to `BodyXY(..., nx=50, ny=50)`. If `sz` is defined along with `nx` or `ny` then a `ValueError` is raised.
- ****kwargs** – Additional arguments are passed to *Body*.

backplanes: `dict[str, Backplane]`

Dictionary containing registered backplanes which can be used to calculate properties (e.g. longitude/latitude, illumination angles etc.) for each pixel in the image.

By default, this dictionary contains a series of *default backplanes*. These can be summarised using *print_backplanes()*. Custom backplanes can be added using *register_backplane()*.

Generated backplane images can be easily retrieved using *get_backplane_img()* and plotted using *plot_backplane_img()*. Similarly, backplane maps cen be retrieved using *get_backplane_map()* and plotted using *plot_backplane_map()*.

This dictionary of backplanes can also be used directly if more customisation is needed:

```
# Retrieve the image containing right ascension values
ra_image = body.backplanes['RA'].get_img()

# Retrieve the map containing emission angles on the target's surface
emission_map = body.backplanes['EMISSION'].get_img()

# Print the description of the doppler factor backplane
print(body.backplanes['DOPPLER'].description)

# Remove the distance backplane from an instance
del body.backplanes['DISTANCE']

# Print summary of all registered backplanes
print(f'{len(body.backplanes)} backplanes currently registered:')
for bp in body.backplanes.values():
    print(f'    {bp.name}: {bp.description}')
```

See *Backplane* for more detail about interacting with the backplanes directly.

Note that a generated backplane image will depend on the disc parameters `(x0, y0, r0, rotation)` at the time the backplane is generated (e.g. calling `body.backplanes['LAT-GRAPHIC'].get_img()` or using *get_backplane_img()*). Generating the same backplane when there are different disc parameter values will produce a different image.

This dictionary is used to define the backplanes saved to the output FITS file in `Observation.save()`.

classmethod from_body(*body:* Body, *nx: int = 0*, *ny: int = 0*, ***, *sz: int | None = None*) → Self

Create a *BodyXY* instance with the same parameters as a *Body* instance.

Parameters

- **body** – *Body* instance to convert.
- **nx** – Number of pixels in the x dimension of the image.
- **ny** – Number of pixels in the y dimension of the image.
- **sz** – Convenience parameter to set both `nx` and `ny` to the same value.

Returns

BodyXY instance with the same parameters as the input *Body* instance and the specified image dimensions.

to_body() → *Body*

Create a *Body* instance from this *BodyXY* instance.

Returns

Body instance with the same parameters as this *BodyXY* instance.

xy2radec(*x: float*, *y: float*) → tuple[float, float]

Convert image pixel coordinates to RA/Dec sky coordinates.

Parameters

- **x** – Image pixel coordinate in the x direction.
- **y** – Image pixel coordinate in the y direction.

Returns

`(ra, dec)` tuple containing the RA/Dec coordinates of the point.

radec2xy(*ra: float*, *dec: float*) → tuple[float, float]

Convert RA/Dec sky coordinates to image pixel coordinates.

Parameters

- **ra** – Right ascension of point in the sky of the observer
- **dec** – Declination of point in the sky of the observer.

Returns

`(x, y)` tuple containing the image pixel coordinates of the point.

xy2lonlat(*x: float*, *y: float*, *not_found_nan=True*) → tuple[float, float]

Convert image pixel coordinates to longitude/latitude coordinates on the target body.

Parameters

- **x** – Image pixel coordinate in the x direction.
- **y** – Image pixel coordinate in the y direction.
- **not_found_nan** – Controls the behaviour when the input `x` and `y` coordinates are missing the target body.

Returns

`(lon, lat)` tuple containing the longitude and latitude of the point. If the provided pixel coordinates are missing the target body, and `not_found_nan` is `True`, then the `lon` and `lat` values will both be NaN.

Raises

NotFoundError – if the input `x` and `y` coordinates are missing the target body and `not_found_nan` is `False`.

lonlat2xy(*lon: float*, *lat: float*) → tuple[float, float]

Convert longitude/latitude on the target body to image pixel coordinates.

Parameters

- **lon** – Longitude of point on target body.
- **lat** – Latitude of point on target body.

Returns

`(x, y)` tuple containing the image pixel coordinates of the point.

xy2km(*x: float*, *y: float*) → tuple[float, float]

Convert image pixel coordinates to distances in the target plane.

Parameters

- **x** – Image pixel coordinate in the x direction.
- **y** – Image pixel coordinate in the y direction.

Returns

`(km_x, km_y)` tuple containing distances in km in the target plane in the East-West and North-South directions respectively.

km2xy(*km_x: float*, *km_y: float*) → tuple[float, float]

Convert distances in the target plane to image pixel coordinates.

Parameters

- **km_x** – Distance in target plane in km in the East-West direction.
- **km_y** – Distance in target plane in km in the North-South direction.

Returns

`(x, y)` tuple containing the image pixel coordinates of the point.

xy2angular(*x: float*, *y: float*, ***angular_kwargs: Unpack[*AngularCoordinateKwargs*]*) → tuple[float, float]

Convert image pixel coordinates to relative angular coordinates.

Parameters

- **x** – Image pixel coordinate in the x direction.
- **y** – Image pixel coordinate in the y direction.
- ****angular_kwargs** – Additional arguments are used to customise the origin and rotation of the relative angular coordinates. See *Body.radec2angular()* for details.

Returns

`(angular_x, angular_y)` tuple containing the relative angular coordinates of the point in arcseconds.

angular2xy(*angular_x: float*, *angular_y: float*, ***angular_kwargs: Unpack[*AngularCoordinateKwargs*]*) → tuple[float, float]

Convert relative angular coordinates to image pixel coordinates.

Parameters

- **angular_x** – Angular coordinate in the x direction in arcseconds.

- **angular_y** – Angular coordinate in the y direction in arcseconds.
- ****angular_kwargs** – Additional arguments are used to customise the origin and rotation of the relative angular coordinates. See *Body.radec2angular()* for details.

Returns
`(x, y)` tuple containing the image pixel coordinates of the point.

set_disc_params(*x0: float | None = None*, *y0: float | None = None*, *r0: float | None = None*, *rotation: float | None = None*) → None

Convenience function to set multiple disc parameters at once.

For example, `body.set_disc_params(x0=10, r0=5)` is equivalent to calling `body.set_x0(10)` and `body.set_r0(5)`. Any unspecified parameters will be left unchanged.

Parameters

- **x0** – If specified, passed to *set_x0()*.
- **y0** – If specified, passed to *set_y0()*.
- **r0** – If specified, passed to *set_r0()*.
- **rotation** – If specified, passed to *set_rotation()*.

adjust_disc_params(*dx: float = 0*, *dy: float = 0*, *dr: float = 0*, *drotation: float = 0*) → None

Convenience function to adjust disc parameters.

This can be used to easily add an offset to disc parameter values without having to call multiple `set_...` and `get_...` functions. For example,

```
body.adjust_disc_params(dy=-3.1, drotation=42)
```

is equivalent to

```
body.set_y0(body.get_y0() - 3.1)
body.set_rotation(body.get_rotation() + 42)
```

The default values of all the arguments are zero, so any unspecified values (e.g. `dx` and `dr` in the example above) are unchanged.

Parameters

- **dx** – Adjustment to `x0`.
- **dy** – Adjustment to `y0`.
- **dr** – Adjustment to `r0`.
- **drotation** – Adjustment to `rotation`.

get_disc_params() → tuple[float, float, float, float]

Convenience function to get all disc parameters at once.

Returns
`(x0, y0, r0, rotation)` tuple.

centre_disc() → None

Centre the target's planetary disc and make it fill ~90% of the observation.

This adjusts `x0` and `y0` so that they lie in the centre of the image, and `r0` is adjusted so that the disc fills 90% of the shortest side of the image. For example, if `nx = 20` and `ny = 30`, then `x0` will be set to 10, `y0` will be set to 15 and `r0` will be set to 9. The rotation of the disc is unchanged.

set_x0(*x0: float*) → None

Parameters
x0 – New x pixel coordinate of the centre of the target body.

Raises
ValueError – if x0 is not finite.

get_x0() → float

Returns
x pixel coordinate of the centre of the target body.

set_y0(*y0: float*) → None

Parameters
y0 – New y pixel coordinate of the centre of the target body.

Raises
ValueError – if y0 is not finite.

get_y0() → float

Returns
y pixel coordinate of the centre of the target body.

set_r0(*r0: float*) → None

Parameters
r0 – New equatorial radius in pixels of the target body.

Raises
ValueError – if r0 is not greater than zero or r0 is not finite.

get_r0() → float

Returns
Equatorial radius in pixels of the target body.

set_rotation(*rotation: float*) → None

Set the rotation of the target body.

This rotation defines the angle between the upwards (positive dec) direction in the RA/Dec sky coordinates and the upwards (positive y) direction in the image pixel coordinates.

Parameters
rotation – New rotation of the target body.

Raises
ValueError – if rotation is not finite.

get_rotation() → float

Returns
Rotation of the target body.

set_plate_scale_arcsec(*arcsec_per_px: float*) → None

Sets the angular plate scale of the observation by changing r0.

Parameters
arcsec_per_px – Arcseconds per pixel plate scale.

`set_plate_scale_km`(*km_per_px: float*) → None

Sets the plate scale of the observation by changing `r0`.

Parameters
km_per_px – Kilometres per pixel plate scale at the target body.

`get_plate_scale_arcsec`() → float

Returns
Plate scale of the observation in arcseconds/pixel.

`get_plate_scale_km`() → float

Returns
Plate scale of the observation in km/pixel at the target body.

`set_img_size`(*nx: int | None = None*, *ny: int | None = None*) → None

Set the `nx` and `ny` values which specify the number of pixels in the x and y dimension of the image respectively. Unspecified values will remain unchanged.

Parameters

- **nx** – If specified, set the number of pixels in the x dimension.
- **ny** – If specified, set the number of pixels in the y dimension.

Raises
`TypeError` – if `set_img_size` is called on an *Observation* instance.

`get_img_size`() → tuple[int, int]

Get the size of the image in pixels.

Returns
`(nx, ny)` tuple containing the number of pixels in the x and y dimension of the image respectively

`set_disc_method`(*method: str*) → None

Record the method used to find the coordinates of the target body's disc. This recorded method can then be used when metadata is saved, such as in `Observation.save()`.

`set_disc_method` is called automatically by functions which find the disc, such as *set_x0()* and `Observation.centre_disc()`, so does not normally need to be manually called by the user.

Parameters
method – Short string describing the method used to find the disc.

`get_disc_method`() → str

Retrieve the method used to find the coordinates of the target body's disc.

Returns
Short string describing the method.

`add_arcsec_offset`(*dra_arcsec: float = 0*, *ddec_arcsec: float = 0*) → None

Adjust the disc location `(x0, y0)` by applying offsets in arcseconds to the RA/Dec celestial coordinates.

Parameters

- **dra_arcsec** – Offset in arcseconds in the positive right ascension direction.
- **ddec_arcsec** – Offset in arcseconds in the positive declination direction.

get_img_limits_radec() → tuple[tuple[float, float], tuple[float, float]]

Get the limits of the image coordinates in the RA/Dec coordinate system.

This can be used to set the axis limits of a plot, for example:

```
xlim, ylim = obs.get_img_limits_radec()
plt.xlim(*xlim)
plt.ylim(*ylim)
```

See also *get_img_limits_km()* and *get_img_limits_xy()*.

Returns
`(ra_left, ra_right), (dec_min, dec_max)` tuple containing the minimum and maximum RA and Dec coordinates of the pixels in the image respectively.

get_img_limits_km() → tuple[tuple[float, float], tuple[float, float]]

Get the limits of the image coordinates in the target centred plane. See *get_img_limits_radec()* for more details.

Returns
`(km_x_min, km_x_max), (km_y_min, km_y_max)` tuple containing the minimum and maximum target plane distance coordinates of the pixels in the image.

get_img_limits_xy() → tuple[tuple[float, float], tuple[float, float]]

Get the limits of the image coordinates. See *get_img_limits_radec()* for more details.

Returns
`(x_min, x_max), (y_min, y_max)` tuple containing the minimum and maximum pixel coordinates of the pixels in the image.

limb_xy(***kwargs*) → tuple[numpy.ndarray, numpy.ndarray]

Pixel coordinate version of *Body.limb_radec()*.

Parameters
****kwargs** – Passed to *Body.limb_radec()*.

Returns
`(x, y)` tuple of coordinate arrays.

limb_xy_by_illumination(***kwargs*) → tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray]

Pixel coordinate version of *Body.limb_radec_by_illumination()*.

Parameters
****kwargs** – Passed to *Body.limb_radec_by_illumination()*.

Returns
`(x_day, y_day, x_night, y_night)` tuple of coordinate arrays of the dayside then nightside parts of the limb.

terminator_xy(***kwargs*) → tuple[numpy.ndarray, numpy.ndarray]

Pixel coordinate version of *Body.terminator_radec()*.

Parameters
****kwargs** – Passed to *Body.terminator_radec()*.

Returns
`(x, y)` tuple of coordinate arrays.

visible_lonlat_grid_xy(**args*, ***kwargs*) → list[tuple[numpy.ndarray, numpy.ndarray]]

Pixel coordinate version of *Body.visible_lonlat_grid_radec()*.

Parameters

- ***args** – Passed to *Body.visible_lonlat_grid_radec()*.
- ****kwargs** – Passed to *Body.visible_lonlat_grid_radec()*.

Returns

List of `(x, y)` coordinate array tuples.

ring_xy(*radius: float*, ***kwargs*) → tuple[numpy.ndarray, numpy.ndarray]

Pixel coordinate version of *Body.ring_radec()*.

Parameters

- **radius** – Radius in km of the ring from the centre of the target body.
- ****kwargs** – Passed to *Body.ring_radec()*.

Returns

`(x, y)` tuple of coordinate arrays.

matplotlib_xy2radec_transform(*ax: matplotlib.axes._axes.Axes | None = None*) → Transform

Get matplotlib transform which converts between coordinate systems.

Transformations to/from the `xy` coordinate system are mutable objects which can be dynamically updated using *update_transform()* when the `radec` to `xy` coordinate conversion changes. This can be useful for plotting data (e.g. an observed image) using image xy coordinates onto an axis using RA/Dec coordinates.

```
# Plot an observed image on an RA/Dec axis with a wireframe of the target
ax = obs.plot_wireframe_radec()
ax.autoscale_view()
ax.autoscale(False) # Prevent imshow breaking autoscale
ax.imshow(
    img,
    origin='lower',
    transform=obs.matplotlib_xy2radec_transform(ax),
    )
```

See *Body.matplotlib_radec2km_transform()* for more details and notes on limitations of these linear transformations.

matplotlib_radec2xy_transform(*ax: matplotlib.axes._axes.Axes | None = None*) → Transform

matplotlib_xy2km_transform(*ax: matplotlib.axes._axes.Axes | None = None*) → Transform

matplotlib_km2xy_transform(*ax: matplotlib.axes._axes.Axes | None = None*) → Transform

matplotlib_xy2angular_transform(*ax: matplotlib.axes._axes.Axes | None = None*, ***angular_kwargs: Unpack[AngularCoordinateKwargs]*) → Transform

matplotlib_angular2xy_transform(*ax: matplotlib.axes._axes.Axes | None = None*, ***angular_kwargs: Unpack[AngularCoordinateKwargs]*) → Transform

update_transform() → None

Update the matplotlib transformations involving `xy` coordinates (e.g. *matplotlib_radec2xy_transform()*) to use the latest disc parameter values `(x0, y0, r0, rotation)`.

map_img(*img: ndarray*, *, *interpolation: Union[Literal['nearest', 'linear', 'quadratic', 'cubic'], int, tuple[int, int]] = 'linear'*, *spline_smoothing: float = 0*, *propagate_nan: bool = True*, *warn_nan: bool = False*, ***map_kwargs: Unpack[MapKwargs]*) → ndarray

Project an observed image to a map. See *generate_map_coordinates()* for details about customising the projection used.

If `interpolation` is `'linear'`, `'quadratic'` or `'cubic'`, the map projection is performed using `scipy.interpolate.RectBivariateSpline` using the specified degree of interpolation.

If `interpolation` is `'nearest'`, no interpolation is performed, and the mapped image takes the value of the nearest pixel in the image to that location. This can be useful to easily visualise the pixel scale for low spatial resolution observations.

To map a cube, this function can be called repeatedly on each image in the cube:

```
mapped_cube = np.array([body.map_img(img) for img in cube])
```

See also *Observation.get_mapped_data()*.

Parameters

- **img** – Observed image where pixel coordinates correspond to the `xy` pixel coordinates (e.g. those used in *get_x0()*).
- **degree_interval** – Interval in degrees between the longitude/latitude points in the mapped output. Passed to *get_x_map()* and *get_y_map()* when generating the coordinates used for the projection.
- **interpolation** – Interpolation used when mapping. This can be any of `'nearest'`, `'linear'`, `'quadratic'` or `'cubic'`; the default is `'linear'`. `'linear'`, `'quadratic'` and `'cubic'` are aliases for spline interpolations of degree 1, 2 and 3 respectively. Alternatively, the degree of spline interpolation can be specified manually by passing an integer or tuple of integers. If an integer is passed, the same interpolation is used in both the x and y directions (i.e. `RectBivariateSpline` with `kx = ky = interpolation`). If a tuple of integers is passed, the first integer is used for the x direction and the second integer is used for the y direction (i.e. `RectBivariateSpline` with `kx, ky = interpolation`).
- **spline_smoothing** – Smoothing factor passed to `RectBivariateSpline(..., s=spline_smoothing)` when spline interpolation is used. This parameter is ignored when `interpolation='nearest'`.
- **propagate_nan** – If using spline interpolation, propagate NaN values from the image to the mapped data. If `propagate_nan` is `True` (the default), the interpolation is performed as normal (i.e. with NaN values in the image set to 0), then any mapped locations where the nearest corresponding image pixel is NaN are set to NaN. Note that there may still be very small errors on the boundaries of NaN regions caused by the interpolation.
- **warn_nan** – Print warning if any values in `img` are NaN when any of the spline interpolations are used.
- ****map_kwargs** – Additional arguments are passed to *generate_map_coordinates()* to specify and customise the map projection.

Returns

Array containing map of the values in `img` at each location on the surface of the target body. Locations which are not visible or outside the projection domain have a value of NaN.

plot_wireframe_xy(*ax: matplotlib.axes._axes.Axes | None = None*, ***, *scale_factor: float | None = None*, *add_axis_labels: bool | None = None*, *aspect_adjustable: Optional[Literal['box', 'datalim']] = 'box'*, *show: bool = False*, ***wireframe_kwargs: Unpack[WireframeKwargs]*) → Axes

Plot basic wireframe representation of the observation using image pixel coordinates. See *Body.plot_wireframe_radec()* for details of accepted arguments.

Returns

The axis containing the plotted wireframe.

plot_map_wireframe(*ax: matplotlib.axes._axes.Axes | None = None*, ***, *label_poles: bool = True*, *add_title: bool = True*, *add_axis_labels: bool = True*, *grid_interval: float = 30*, *grid_lat_limit: float = 90*, *indicate_equator: bool = True*, *indicate_prime_meridian: bool = True*, *aspect_adjustable: Optional[Literal['box', 'datalim']] = 'box'*, *formatting: dict[Literal['all', 'grid', 'equator', 'prime_meridian', 'limb', 'limb_illuminated', 'terminator', 'ring', 'pole', 'coordinate_of_interest_lonlat', 'coordinate_of_interest_radec', 'other_body_of_interest_marker', 'other_body_of_interest_label', 'hidden_other_body_of_interest_marker', 'hidden_other_body_of_interest_label', 'map_boundary'], dict[str, Any]] | None = None*, ***map_and_formatting_kwargs*) → Axes

Plot wireframe (e.g. gridlines) of the map projection of the observation. See *Body.plot_wireframe_radec()* for details of accepted arguments.

For example, to plot an orthographic map's wireframe with a red boundary and dashed gridlines, you can use:

```
body.plot_map_wireframe(
    projection='orthographic',
    lat=45,
    formatting={
        'grid': {'linestyle': '--'},
        'map_boundary': {'color': 'red'},
    }
)
```

plot_map(*map_img: ndarray*, *ax: matplotlib.axes._axes.Axes | None = None*, ***, *wireframe_kwargs: dict[str, Any] | None = None*, *add_wireframe: bool = True*, ***kwargs*) → QuadMesh

Utility function to easily plot a mapped image using `plt.imshow` with appropriate extents, axis labels, gridlines etc.

Parameters

- **map_img** – Image to plot.
- **ax** – Matplotlib axis to use for plotting. If `ax` is None (the default), then a new figure and axis is created.
- **wireframe_kwargs** – Dictionary of arguments passed to *plot_map_wireframe()*.
- **add_wireframe** – Enable/disable plotting of wireframe.
- ****kwargs** – Additional arguments are passed to *generate_map_coordinates()* to specify the map projection used, and to Matplotlib's `pcolormesh` to customise the plot. For example, can be used to set the colormap of the plot using e.g. `body.plot_map(..., projection='orthographic', cmap='Greys')`.

Returns

Handle returned by Matplotlib's `pcolormesh`.

get_wireframe_overlay_img(*output_size: int | None = 1500*, *dpi: int = 200*, *rgba: bool = False*, ***plot_kwargs*) → ndarray

Warning: This is a beta feature and the API may change in future.

Generate a wireframe image of the target.

This effectively generates an image version of *plot_wireframe_xy()* which can then be used as an overlay on top of the observation when creating figures in other applications.

See also *get_wireframe_overlay_map()*.

Note: The returned image data follows the FITS orientation convention (with the origin at the bottom left) so may need to be flipped vertically in some applications. If needed, the image can be flipped in Python using:

```
np.flipud(body.get_wireframe_overlay_img())
```

Hint: If you are creating plots with Matplotlib, it is generally better to use *plot_wireframe_xy()* directly rather than generating an image as it will produce a higher quality plot.

Parameters

- **output_size** – Size of the output image in pixels. This will be the length of the longest side of the image. The other side will be scaled accordingly to maintain the aspect ratio of the observed data. If `size` is `None`, then the size is set to match the size of the observed data.
- **dpi** – Dots per inch of the output image. This can be used to control the size of plotted elements in the output image - larger `dpi` values will produce larger plotted elements.
- **rgba** – By default, the returned image only has a single greyscale channel. If `rgba` is `True`, then the returned image has 4 channels (red, green, blue, alpha) which can be used to more easily overlay the wireframe on top of the observed data in other applications.
- ****plot_kwargs** – Additional arguments passed to *plot_wireframe_xy()*.

Returns

Image of the wireframe which has the same aspect ratio as the observed data.

get_wireframe_overlay_map(*output_size: int | None = 1500*, *dpi: int = 200*, *rgba: bool = False*, ***map_and_formatting_kwargs*) → ndarray

Warning: This is a beta feature and the API may change in future.

Generate a wireframe map of the target.

This effectively generates an image version of *plot_map_wireframe()* which can then be used as an overlay on top of the mapped observation when creating figures in other applications.

See also *get_wireframe_overlay_img()*.

Note: The returned image data follows the FITS orientation convention (with the origin at the bottom left) so may need to be flipped vertically in some applications. If needed, the image can be flipped in Python using:

```
np.flipud(body.get_wireframe_overlay_map())
```

Hint: If you are creating plots with Matplotlib, it is generally better to use *plot_map_wireframe()* directly rather than generating an image as it will produce a higher quality plot.

Parameters

- **output_size** – Size of the output image in pixels. This will be the length of the longest side of the map. The other side will be scaled accordingly to maintain the aspect ratio of the observed data. If `size` is `None`, then the size is set to match the pixel size of the map.
- **dpi** – Dots per inch of the output image. This can be used to control the size of plotted elements in the output image - larger `dpi` values will produce larger plotted elements.
- **rgba** – By default, the returned image only has a single greyscale channel. If `rgba` is `True`, then the returned image has 4 channels (red, green, blue, alpha) which can be used to more easily overlay the wireframe on top of the observed data in other applications.
- **plot_kwargs** – Dictionary of arguments passed to *plot_map_wireframe()*.
- ****map_and_formatting_kwargs** – Passed to *plot_map_wireframe()*. This can include arguments such as `projection`.

Returns

Image of the map wireframe which has the same aspect ratio as the map.

static standardise_backplane_name(*name: str*) → str

Create a standardised version of a backplane name when finding and registering backplanes.

This standardisation is used in functions like *get_backplane_img()* and `plot_backplane()` so that, for example `body.plot_backplane('DEC')`, `body.plot_backplane('Dec')` and `body.plot_backplane('dec')` all produce the same plot.

Parameters

name – Input backplane name.

Returns

Standardised name with leading/trailing spaces removed and converted to upper case.

register_backplane(*name: str*, *description: str*, *get_img: Callable[[], ndarray]*, *get_map: _BackplaneMapGetter*) → None

Create a new *Backplane* and register it to *backplanes*.

See *Backplane* for more detail about parameters.

Parameters

- **name** – Name of backplane. This is standardised using *standardise_backplane_name()* before being registered.
- **description** – Longer description of backplane, including units.
- **get_img** – Function to generate backplane image.

- **get_map** – Function to generate backplane map.

Raises
ValueError – if provided backplane name is already registered.

backplane_summary_string() → str

Returns
String summarising currently registered *backplanes*.

print_backplanes() → None

Prints output of *backplane_summary_string()*.

get_backplane(*name: str*) → *Backplane*

Convenience function to retrieve a backplane registered to *backplanes*.

This method is equivalent to

```
body.backplanes[self.standardise_backplane_name(name)]
```

Parameters
name – Name of the desired backplane. This is standardised with *standardise_backplane_name()* and used to choose a registered backplane from *backplanes*.

Returns
Backplane registered with `name`.

Raises
BackplaneNotFoundError – if `name` is not registered as a backplane.

get_backplane_img(*name: str*) → ndarray

Generate backplane image.

Note that a generated backplane image will depend on the disc parameters `(x0, y0, r0, rotation)` at the time this function is called. Generating the same backplane when there are different disc parameter values will produce a different image. This method creates a copy of the generated image, so the returned image can be safely modified without affecting the cached value (unlike the return values from functions such as *get_lon_img()*).

This method is equivalent to

```
body.get_backplane(name).get_img().copy()
```

Parameters
name – Name of the desired backplane. This is standardised with *standardise_backplane_name()* and used to choose a registered backplane from *backplanes*.

Returns
Array containing the backplane's values for each pixel in the image.

get_backplane_map(*name: str*, ***map_kwargs: Unpack[*MapKwargs*]*) → ndarray

Generate map of backplane values.

This method creates a copy of the generated image, so the returned map can be safely modified without affecting the cached value (unlike the return values from functions such as *get_lon_map()*).

This method is equivalent to

```
body.get_backplane(name).get_map(degree_interval).copy()
```

Parameters

- **name** – Name of the desired backplane. This is standardised with *standardise_backplane_name()* and used to choose a registered backplane from *backplanes*.
- ****map_kwargs** – Additional arguments are passed to *generate_map_coordinates()* to specify and customise the map projection.

Returns

Array containing map of the backplane's values over the surface of the target body.

plot_backplane_img(*name: str*, *ax: matplotlib.axes._axes.Axes | None = None*, *show: bool = False*, ***kwargs*) → Axes

Plot a backplane image with the wireframe outline of the target.

Note that a generated backplane image will depend on the disc parameters `(x0, y0, r0, rotation)` at the time this function is called. Generating the same backplane when there are different disc parameter values will produce a different image.

Parameters

- **name** – Name of the desired backplane.
- **ax** – Passed to *plot_wireframe_xy()*.
- **show** – Passed to *plot_wireframe_xy()*.
- ****kwargs** – Passed to Matplotlib's `imshow` when plotting the backplane image. For example, can be used to set the colormap of the plot using `body.plot_backplane_img(..., cmap='Greys')`.

Returns

The axis containing the plotted data.

plot_backplane_map(*name: str*, *ax: matplotlib.axes._axes.Axes | None = None*, *show: bool = False*, ***kwargs*) → Axes

Plot a map of backplane values on the target body.

For example, top plot a backplane map with the 'Blues' colourmap and a red partially transparent wireframe, on an orthographic projection, use:

```
body.plot_backplane_map(
    'EMISSION',
    projection='orthographic',
    cmap='Blues',
    wireframe_kwargs=dict(color='r', alpha=0.5),
)
```

Parameters

- **name** – Name of the desired backplane.
- **ax** – Matplotlib axis to use for plotting. If `ax` is None (the default), then a new figure and axis is created.
- **show** – Toggle showing the plotted figure with `plt.show()`

- ****kwargs** – Additional arguments are passed to *plot_map()*. These can be used to specify and customise the map projection, and to customise the plot formatting.

Returns
The axis containing the plotted data.

generate_map_coordinates(*projection: str = 'rectangular'*, *degree_interval: float = 1*, *lon: float = 0*, *lat: float = 0*, *size: int = 100*, *lon_coords: numpy.ndarray | tuple | None = None*, *lat_coords: numpy.ndarray | tuple | None = None*, *projection_x_coords: numpy.ndarray | tuple | None = None*, *projection_y_coords: numpy.ndarray | tuple | None = None*, *xlim: tuple[float, float] | None = None*, *ylim: tuple[float, float] | None = None*) → tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray, pyproj.transformer.Transformer, dict[str, Any]]

Generate underlying coordinates and transformation for a given map projection.

The built-in map projections (i.e. possible values for the `projection` argument) are:

- `'rectangular'`: cylindrical equirectangular projection onto a regular longitude and latitude grid. The resolution of the map can be controlled with the `degree_interval` argument which sets the spacing in degrees between grid points. This is the default map projection.
- `'orthographic'`: orthographic projection where the central longitude and latitude can be customized with the `lon` and `lat` arguments. The size of the map can be controlled with the `size` argument.
- `'azimuthal'`: azimuthal equidistant projection where the central longitude and latitude can be customized with the `lon` and `lat` arguments. The size of the map can be controlled with the `size` argument.
- `'azimuthal equal area'`: Lambert azimuthal equal area projection where the central longitude and latitude can be customized with the `lon` and `lat` arguments. The size of the map can be controlled with the `size` argument.
- `'manual'`: manually specify the longitude and latitude coordinates to use for each point on the map with the `lon_coords` and `lat_coords` arguments.

Projections can also be specified by passing a proj projection string to the `projection` argument. If you are manually specifying a projection, you must also specify `projection_x_coords` and `projection_y_coords` to provide the x and y coordinates to project the data to. See https://proj.org/operations/projections for a list of projections that can be used. The provided projection string will be passed to `pyproj.CRS`. *create_proj_string()* can be used to help build a projection string.

Hint: You generally don't need to call this method directly. Instead, pass your desired arguments directly to functions like *get_backplane_map()* or *map_img()*.

Usage examples:

```
# Generate default rectangular map for emission backplane
body.get_backplane_map('EMISSION')

# Generate default rectangular map at lower resolution and only covering
# the northern hemisphere
body.get_backplane_map('EMISSION', degree_interval=10, ylim=(0, np.inf))

# Generate orthographic map of northern hemisphere
body.get_backplane_map('EMISSION', projection='orthographic', lat=90)
```

(continues on next page)

(continued from previous page)

```
# Plot orthographic map of southern hemisphere with higher resolution
body.plot_backplane_map(
    'EMISSION', projection='orthographic', lat=-90, size=500
    )

# Get azimuthal equidistant map projection of image, centred on specific
# coordinate
body.map_img(img, projection='azimuthal', lon=45, lat=30)
```

Parameters

- **projection** – String describing map projection to use (see list of supported projections above).
- **degree_interval** – Degree interval for `'rectangular` projection.
- **lon** – Central longitude of `'orthographic'`, `'azimuthal'` and `'azimuthal equal area'` projections.
- **lat** – Central latitude of `'orthographic'`, `'azimuthal'` and `'azimuthal equal area'` projections.
- **size** – Pixel size (width and height) of generated `'orthographic'`, `'azimuthal'` and `'azimuthal equal area'` projections.
- **lon_coords** – Longitude coordinates to use for `'manual'` projection. This must be a tuple (e.g. use `lon_coords=tuple(np.linspace(0, 360, 100))`) - this allows mapping arguments and outputs to be cached).
- **lat_coords** – Latitude coordinates to use for `'manual'` projection. This must be a tuple.
- **projection_x_coords** – Projected x coordinates to use with a pyproj projection string. This must be a tuple.
- **projection_y_coords** – Projected x coordinates to use with a pyproj projection string. This must be a tuple.
- **xlim** – Tuple of `(x_min, x_max)` limits in the projected x coordinates of the map. If `None`, the default, then the no limits are applied (i.e. the entire globe will be mapped). If `xlim` is provided, it should be a tuple of two floats specifying the minimum and maximum x coordinates to project the map to. For example, to only plot the western hemisphere, you can use use `xlim=(0, 180)` in a rectangular projection. Note that these limits are expressed in the projected coordinates of the map. Setting the limits can be useful to speed up the performance of mapping when only a subset of the map is needed (such as for observations with limited spatial extent). If you only want to set one limit, then you can pass infinity e.g. `xlim=(315, np.inf)` to only set the minimum limit. The limits are implemented using `x_to_keep = (x >= min(xlim)) & (x <= max(xlim))`, so the ordering of the limits does not matter. Note that the limit calculations assume that the data is on a rectangular grid (i.e. all rows have the same x coordinates and all columns have the same y coordinates), so may produce unexpected results if a custom projection is used.
- **ylim** – Tuple of `(y_min, y_max)` limits in the projected y coordinates of the map. If `None`, the default, then the no limits are applied. See `xlim` for more details.

Returns

`(lons, lats, xx, yy, transformer, info)` tuple where `lons` and `lats` are the longitude and latitude coordinates of the map, `xx` and `yy` are the projected coordinates of the

map, `transformer` is a `pyproj.Transformer` object that can be used to transform between the two coordinate systems, and `info` is a dictionary containing the arguments used to build the map (e.g. for the default case this would be `{'projection': 'rectangular', 'degree_interval': 1, 'xlim': None, 'ylim': None}`).

create_proj_string(*proj: str*, ***parameters*) → str

Create projection string for use with pyproj.

This function will automatically build a projection string that can be used as the `projection` argument of *generate_map_coordinates()*.

By default, this function automatically sets the `+axis` parameter of the projection to match the *Body.positive_longitude_direction* of the target body - if the target body has a positive longitude direction of E, then the projection will have `+axis=enu`, if the target body has a positive longitude direction of W, then the projection will have `+axis=wnu`. This behaviour can be disabled by passing `axis=None` to this function. See https://proj.org/usage/projections.html#axis-orientation for more details about the `+axis` projection parameter.

Examples:

```
body.create_proj_string('ortho')
# '+proj=ortho +axis=wnu +type=crs'

body.create_proj_string('ortho', lon_0=180, lat_0=45)
# '+proj=ortho +lon_0=180 +lat_0=45 +axis=wnu +type=crs'

body.create_proj_string('ortho', lon_0=180, lat_0=45, axis=None)
# '+proj=ortho +lon_0=180 +lat_0=45 +type=crs'
```

Parameters

- **proj** – Projection name. See https://proj.org/operations/projections for a full list of projections that can be used.
- ****parameters** – Additional parameters to pass to the projection. These are passed to pyproj as `+{key}={value}`. For example, to create a projection with a central longitude of 45 degrees, you can use `lon_0=45`. By default, the axis direction is set to match the *Body.positive_longitude_direction* of the target body (see above), pass `axis=None` to disable this behaviour.

Returns

Proj string describing the projection. This can be passed to the `projection` argument of *generate_map_coordinates()*.

get_lon_img() → ndarray

See also *get_backplane_img()*.

Returns

Array containing the planetographic longitude value of each pixel in the image. Points off the disc have a value of NaN.

get_lon_map(***map_kwargs: Unpack[MapKwargs]*) → ndarray

See *generate_map_coordinates()* for accepted arguments. See also *get_backplane_map()*.

Returns

Array containing map of planetographic longitude values.

get_lat_img() → ndarray

See also *get_backplane_img()*.

Returns

Array containing the planetographic latitude value of each pixel in the image. Points off the disc have a value of NaN.

get_lat_map(***map_kwargs: Unpack*[MapKwargs]) → ndarray

See *generate_map_coordinates()* for accepted arguments. See also *get_backplane_map()*.

Returns

Array containing map of planetographic latitude values.

get_lon_centric_img() → ndarray

See also *get_backplane_img()*.

Returns

Array containing the planetocentric longitude value of each pixel in the image. Points off the disc have a value of NaN.

get_lon_centric_map(***map_kwargs: Unpack*[MapKwargs]) → ndarray

See *generate_map_coordinates()* for accepted arguments. See also *get_backplane_map()*.

Returns

Array containing map of planetocentric longitude values.

get_lat_centric_img() → ndarray

See also *get_backplane_img()*.

Returns

Array containing the planetocentric latitude value of each pixel in the image. Points off the disc have a value of NaN.

get_lat_centric_map(***map_kwargs: Unpack*[MapKwargs]) → ndarray

See *generate_map_coordinates()* for accepted arguments. See also *get_backplane_map()*.

Returns

Array containing map of planetocentric latitude values.

get_ra_img() → ndarray

See also *get_backplane_img()*.

Returns

Array containing the right ascension (RA) value of each pixel in the image.

get_ra_map(***map_kwargs: Unpack*[MapKwargs]) → ndarray

See *generate_map_coordinates()* for accepted arguments. See also *get_backplane_map()*.

Returns

Array containing map of right ascension values as viewed by the observer. Locations which are not visible have a value of NaN.

get_dec_img() → ndarray

See also *get_backplane_img()*.

Returns

Array containing the declination (Dec) value of each pixel in the image.

get_dec_map(***map_kwargs: *Unpack*[MapKwargs]*) → ndarray

See *generate_map_coordinates()* for accepted arguments. See also *get_backplane_map()*.

Returns

Array containing map of declination values as viewed by the observer. Locations which are not visible have a value of NaN.

get_x_img() → ndarray

See also *get_backplane_img()*.

Returns

Array containing the x pixel coordinate value of each pixel in the image.

get_x_map(***map_kwargs: *Unpack*[MapKwargs]*) → ndarray

See *generate_map_coordinates()* for accepted arguments. See also *get_backplane_map()*.

Returns

Array containing map of the x pixel coordinates each location corresponds to in the observation. Locations which are not visible or are not in the image frame have a value of NaN.

get_y_img() → ndarray

See also *get_backplane_img()*.

Returns

Array containing the y pixel coordinate value of each pixel in the image.

get_y_map(***map_kwargs: *Unpack*[MapKwargs]*) → ndarray

See *generate_map_coordinates()* for accepted arguments. See also *get_backplane_map()*.

Returns

Array containing map of the y pixel coordinates each location corresponds to in the observation. Locations which are not visible or are not in the image frame have a value of NaN.

get_km_x_img() → ndarray

See also *get_backplane_img()*.

Returns

Array containing the distance in target plane in km in the East-West direction.

get_km_x_map(***map_kwargs: *Unpack*[MapKwargs]*) → ndarray

See *generate_map_coordinates()* for accepted arguments. See also *get_backplane_map()*.

Returns

Array containing map of the distance in target plane in km in the East-West direction. Locations which are not visible have a value of NaN.

get_km_y_img() → ndarray

See also *get_backplane_img()*.

Returns

Array containing the distance in target plane in km in the North-South direction.

get_km_y_map(***map_kwargs: *Unpack*[MapKwargs]*) → ndarray

See *generate_map_coordinates()* for accepted arguments. See also *get_backplane_map()*.

Returns

Array containing map of the distance in target plane in km in the North-South direction. Locations which are not visible have a value of NaN.

get_phase_angle_img() → ndarray

See also *get_backplane_img()*.

Returns

Array containing the phase angle value of each pixel in the image. Points off the disc have a value of NaN.

get_phase_angle_map(**map_kwargs: *Unpack*[MapKwargs]) → ndarray

See *generate_map_coordinates()* for accepted arguments. See also *get_backplane_map()*.

Returns

Array containing map of the phase angle value at each point on the target's surface.

get_incidence_angle_img() → ndarray

See also *get_backplane_img()*.

Returns

Array containing the incidence angle value of each pixel in the image. Points off the disc have a value of NaN.

get_incidence_angle_map(**map_kwargs: *Unpack*[MapKwargs]) → ndarray

See *generate_map_coordinates()* for accepted arguments. See also *get_backplane_map()*.

Returns

Array containing map of the incidence angle value at each point on the target's surface.

get_emission_angle_img() → ndarray

See also *get_backplane_img()*.

Returns

Array containing the emission angle value of each pixel in the image. Points off the disc have a value of NaN.

get_emission_angle_map(**map_kwargs: *Unpack*[MapKwargs]) → ndarray

See *generate_map_coordinates()* for accepted arguments. See also *get_backplane_map()*.

Returns

Array containing map of the emission angle value at each point on the target's surface.

get_azimuth_angle_img() → ndarray

See also *get_backplane_img()*.

Returns

Array containing the azimuth angle value of each pixel in the image. Points off the disc have a value of NaN.

get_azimuth_angle_map(**map_kwargs: *Unpack*[MapKwargs]) → ndarray

See *generate_map_coordinates()* for accepted arguments. See also *get_backplane_map()*.

Returns

Array containing map of the azimuth angle value at each point on the target's surface.

get_local_solar_time_img() → ndarray

See also *get_backplane_img()*.

Returns

Array containing the local solar time value of each pixel in the image, as calculated by *Body.local_solar_time_from_lon()*. Points off the disc have a value of NaN.

get_local_solar_time_map(**map_kwargs: *Unpack*[MapKwargs]) → ndarray

See *generate_map_coordinates()* for accepted arguments. See also *get_backplane_map()*.

Returns

Array containing map of the local solar time at each point on the target's surface, as calculated by *Body.local_solar_time_from_lon()*.

get_distance_img() → ndarray

See also *get_backplane_img()*.

Returns

Array containing the observer-target distance in km of each pixel in the image. Points off the disc have a value of NaN.

get_distance_map(**map_kwargs: *Unpack*[MapKwargs]) → ndarray

See *generate_map_coordinates()* for accepted arguments. See also *get_backplane_map()*.

Returns

Array containing map of the observer-target distance in km of each point on the target's surface.

get_radial_velocity_img() → ndarray

See also *get_backplane_img()*.

Returns

Array containing the observer-target radial velocity in km/s of each pixel in the image. Velocities towards the observer are negative. Points off the disc have a value of NaN.

get_radial_velocity_map(**map_kwargs: *Unpack*[MapKwargs]) → ndarray

See *generate_map_coordinates()* for accepted arguments. See also *get_backplane_map()*.

Returns

Array containing map of the observer-target radial velocity in km/s of each point on the target's surface.

get_doppler_img() → ndarray

See also *get_backplane_img()*.

Returns

Array containing the doppler factor for each pixel in the image, calculated using *SpiceBase.calculate_doppler_factor()* on velocities from *get_radial_velocity_img()*. Points off the disc have a value of NaN.

get_doppler_map(**map_kwargs: *Unpack*[MapKwargs]) → ndarray

See *generate_map_coordinates()* for accepted arguments. See also *get_backplane_map()*.

Returns

Array containing map of the doppler factor of each point on the target's surface. This is calculated using *SpiceBase.calculate_doppler_factor()* on velocities from *get_radial_velocity_map()*.

get_limb_lon_img() → ndarray

See also *get_backplane_img()*.

Returns

Array containing the planetographic longitude of the point on the target's limb that is closest to each pixel. See *Body.limb_coordinates_from_radec()* for more detail.

get_limb_lon_map(**map_kwargs: *Unpack*[MapKwargs]) → ndarray

See *generate_map_coordinates()* for accepted arguments. See also *get_backplane_map()*.

Returns

Array containing map of the planetographic longitude of the point on the target's limb that is closest to each point on the target's surface (for the observer). See *Body.limb_coordinates_from_radec()* for more detail.

get_limb_lat_img() → ndarray

See also *get_backplane_img()*.

Returns

Array containing the planetographic latitude of the point on the target's limb that is closest to each pixel. See *Body.limb_coordinates_from_radec()* for more detail.

get_limb_lat_map(**map_kwargs: *Unpack*[MapKwargs]) → ndarray

See *generate_map_coordinates()* for accepted arguments. See also *get_backplane_map()*.

Returns

Array containing map of the planetographic latitude of the point on the target's limb that is closest to each point on the target's surface (for the observer). See *Body.limb_coordinates_from_radec()* for more detail.

get_limb_distance_img() → ndarray

See also *get_backplane_img()*.

Returns

Array containing the distance in km above the target's limb for each pixel. See *Body.limb_coordinates_from_radec()* for more detail.

get_limb_distance_map(**map_kwargs: *Unpack*[MapKwargs]) → ndarray

See *generate_map_coordinates()* for accepted arguments. See also *get_backplane_map()*.

Returns

Array containing map of the distance in km above the target's limb for each point on the target's surface (for the observer). See *Body.limb_coordinates_from_radec()* for more detail.

get_ring_plane_radius_img() → ndarray

See also *get_backplane_img()*.

Returns

Array containing the ring plane radius in km for each pixel in the image, calculated using *Body.ring_plane_coordinates()*. Points of the ring plane obscured by the target body have a value of NaN.

get_ring_plane_radius_map(**map_kwargs: *Unpack*[MapKwargs]) → ndarray

See *generate_map_coordinates()* for accepted arguments. See also *get_backplane_map()*.

Returns

Array containing map of the ring plane radius in km obscuring each point on the target's surface, calculated using *Body.ring_plane_coordinates()*. Points where the target body is unobscured by the ring plane have a value of NaN.

get_ring_plane_longitude_img() → ndarray

See also *get_backplane_img()*.

Returns

Array containing the ring plane planetographic longitude in degrees for each pixel in the

image, calculated using *Body.ring_plane_coordinates()*. Points of the ring plane obscured by the target body have a value of NaN.

get_ring_plane_longitude_map(***map_kwargs: Unpack[MapKwargs]*) → ndarray

See *generate_map_coordinates()* for accepted arguments. See also *get_backplane_map()*.

Returns

Array containing map of the ring plane planetographic longitude in degrees obscuring each point on the target's surface, calculated using *Body.ring_plane_coordinates()*. Points where the target body is unobscured by the ring plane have a value of NaN.

get_ring_plane_distance_img() → ndarray

See also *get_backplane_img()*.

Returns

Array containing the ring plane distance from the observer in km for each pixel in the image, calculated using *Body.ring_plane_coordinates()*. Points of the ring plane obscured by the target body have a value of NaN.

get_ring_plane_distance_map(***map_kwargs: Unpack[MapKwargs]*) → ndarray

See *generate_map_coordinates()* for accepted arguments. See also *get_backplane_map()*.

Returns

Array containing map of the ring plane distance from the observer in km obscuring each point on the target's surface, calculated using *Body.ring_plane_coordinates()*. Points where the target body is unobscured by the ring plane have a value of NaN.

class planetmapper.**Observation**(*path: str | os.PathLike | None = None*, ***, *data: numpy.ndarray | None = None*, *header: astropy.io.fits.header.Header | None = None*, ***kwargs*)

Bases: *BodyXY*

Class representing an actual observation of an astronomical body at a specific time.

This is a subclass of *BodyXY*, with additional methods to interact with the observed data, such as by saving a FITS file containing calculated backplane data. All methods described in *BodyXY*, *Body* and *SpiceBase* are therefore available in instances of this class.

This class can be created by either providing a `path` to a data file to be loaded, or by directly providing the `data` itself (and optionally a FITS `header`). The `nx` and `ny` values for *BodyXY* will automatically be calculated from the input data.

If the input data is a FITS file (or a `header` is specified), then this class will attempt to generate appropriate parameters (e.g. `target`, `utc`) by using the values in the header. This allows an instance of this class to be created with a single argument specifying the `path` to the FITS file e.g. `Observation('path/to/file.fits')`. Manually specified parameters will take precedence, so `Observation('path/to/file.fits', target='JUPITER')` will have Jupiter as a target, regardless of any values saying otherwise in the FITS header.

If a FITS header is not provided (e.g. if the input path corresponds to an image file), then at least the `target` and `utc` parameters need to be specified.

When an `Observation` object is created, the disc parameters `(x0, y0, r0, rotation)` initialised to the most useful values possible:

1. If the input file has previously been fit by PlanetMapper, the previous parameter values saved in the FITS header are loaded using *disc_from_header()*.
2. Otherwise, if there is WCS information in the FITS header, this is loaded with *disc_from_wcs()*.
3. Finally, if there is no useful information in the FITS header (or no header is provided), the disc parameters are initialised using `centre_disc()`.

Parameters

- **path** – Path to data file to load. If this is `None` then `data` must be specified instead. Any user (~) and shell variables (e.g. `$var`) in the path are automatically expanded if possible.
- **data** – Array containing observation data to use instead of loading the data from `path`. This should only be provided if `path` is None.
- **header** – FITS header which corresponds to the provided `data`. This is optional and should only be provided if `path` is None.
- **target** – Name of target body, passed to *Body*. If this is unspecified, then the target will be derived from the values in the FITS header.
- **utc** – Time of observation, passed to *Body*. If this is unspecified, then the time will be derived from the values in the FITS header.
- ****kwargs** – Additional parameters are passed to *BodyXY*. These can be used to specify additional parameters such as`observer`. The image size is automatically determined from the data, so passing `nx`, `ny` or `sz` as arguments when creating an `Observation` object will raise a `TypeError`.

FITS_FILE_EXTENSIONS = ('.fits', '.fits.gz')

File extensions which will be read as FITS files. All other file extensions will be assumed to be images.

FITS_KEYWORD = 'PLANMAP'

FITS keyword used in metadata added to header of output FITS files.

path: str | None

Path of input data file, or `None` if no file was provided.

data: np.ndarray

Observed data.

header: fits.Header

FITS header containing data about the observation. If this is not provided, then a basic header will be produced containing data derived from the `target` and `utc` parameters.

to_body_xy() → *BodyXY*

Create a *BodyXY* object with the same parameters and data as this observation.

Returns

BodyXY object with the same disc parameters as this *Observation* instance.

disc_from_header() → None

Sets the target's planetary disc data in the FITS header generated by previous runs of `planetmapper`.

This uses values such as `HIERARCH PLANMAP DISC X0` to set the disc location to be the same as the previous run.

Raises

ValueError – if the header does not contain appropriate metadata values. This is likely because the file was not created by `planetmapper`.

disc_from_wcs(*suppress_warnings: bool = False*, *validate: bool = True*, *use_header_offsets: bool = True*) → None

Set disc parameters using WCS information in the observation's FITS header.

See also *rotation_from_wcs()* and *plate_scale_from_wcs()*.

Note: There may be very slight differences between the coordinates converted directly from the WCS information and the coordinates converted by PlanetMapper.

Parameters

- **suppress_warnings** – Hide warnings produced by astropy when calculating WCS conversions.
- **validate** – Run checks to ensure the WCS conversion has appropriate RA/Dec coordinate dimensions.
- **use_header_offsets** – If present, use the `HIERARCH NAV RA_OFFSET` and `HIERARCH NAV DEC_OFFSET` values from the FITS headerr to adjust the target's disc location by the specified arcsecond offsets. If these keywords are not present or `use_header_offsets` is `False`, no adjustment is made.

Raises

`ValueError` – if no WCS information is found in the FITS header, or validation fails.

position_from_wcs(**args*, ***kwargs*) → None

Set disc position `(x0, y0)` using WCS information in the observation's FITS header.

See also *disc_from_wcs()*.

Parameters

- ***args** – See *disc_from_wcs()* for additional arguments.
- ****kwargs** – See *disc_from_wcs()* for additional arguments.

Raises

`ValueError` – if no WCS information is found in the FITS header, or validation fails.

rotation_from_wcs(**args*, ***kwargs*) → None

Set disc rotation using WCS information in the observation's FITS header.

See also *disc_from_wcs()*.

Parameters

- ***args** – See *disc_from_wcs()* for additional arguments.
- ****kwargs** – See *disc_from_wcs()* for additional arguments.

Raises

`ValueError` – if no WCS information is found in the FITS header, or validation fails.

plate_scale_from_wcs(**args*, ***kwargs*) → None

Set plate scale (i.e. `r0`) using WCS information in the observation's FITS header.

See also *disc_from_wcs()*.

Parameters

- ***args** – See *disc_from_wcs()* for additional arguments.
- ****kwargs** – See *disc_from_wcs()* for additional arguments.

Raises

`ValueError` – if no WCS information is found in the FITS header, or validation fails.

get_wcs_offset(**args*, ***kwargs*) → tuple[float, float, float, float]

> **Warning:** This is a beta feature and the API may change in future.

Get the difference between the current disc parameters and the disc parameters calculated from the WCS information in the observation's FITS header.

For example, this function can be used to retrieve the cumulative offset after adjusting the disc position:

```
# Initialise disc with parameters from WCS
observation.disc_from_wcs()

# Adjust the disc position
observation.adjust_disc_params(1, 2, 3, 4)
observation.adjust_disc_params(dx=0.1)

# Retrieve the cumulative offset
print(observation.get_wcs_offset())  # (1.1, 2.0, 3.0, 4.0)
```

Similarly, this function can be used to retrieve the offset after running the GUI to fit the disc:

```
observation.run_gui()
print(observation.get_wcs_offset())
```

See also *get_wcs_arcsec_offset()*.

Parameters

- ***args** – See *disc_from_wcs()* for additional arguments.
- ****kwargs** – See *disc_from_wcs()* for additional arguments.

Returns

`(dx, dy, dr, drotation)` tuple containing the differences in disc parameters between the current disc parameters (i.e. those returned by *BodyXY.get_disc_params()*) and the disc parameters calculated from the WCS information in the observation's FITS header. `dx` and `dy` give the difference in the disc centre position in pixels, `dr` gives the difference in the disc radius in pixels, and `drotation` gives the difference in the rotation angle in degrees.

Raises

ValueError – if no WCS information is found in the FITS header, or validation fails.

get_wcs_arcsec_offset(**args*, *check_is_position_offset_only: bool = True*, ***kwargs*) → tuple[float, float]

> **Warning:** This is a beta feature and the API may change in future.

Get the offset in RA/Dec celestial coordinates between the current disc location and the disc location calculated from the WCS information in the observation's FITS header.

For example, this function can be used to retrieve the cumulative offset after adjusting the disc position:

```
# Initialise disc with parameters from WCS
observation.disc_from_wcs()
```

(continues on next page)

(continued from previous page)

```
# Adjust the disc position
observation.add_arcsec_offset(10, 10)
observation.add_arcsec_offset(dra_arcsec=1.23)

# Retrieve the cumulative offset
print(observation.get_wcs_arcsec_offset())  # (11.23, 10.0)
```

Similarly, this function can be used to retrieve the offset after running the GUI to fit the disc:

```
observation.run_gui()
print(observation.get_wcs_arcsec_offset())
```

The RA/Dec offsets returned by this function are generally only meaningful if the disc location `(x0, y0)` is the only difference between the current disc parameters and those derived from the WCS. Therefore, by default this function checks that the `dr` and `drotation` values returned by *get_wcs_offset()* are sufficiently small to be considered a position offset only, and raises a `ValueError` if this is not the case. This check can be disabled by setting `check_is_position_offset_only` to `False`.

See also *get_wcs_offset()*.

Parameters

- ***args** – See *disc_from_wcs()* for additional arguments.
- ****kwargs** – See *disc_from_wcs()* for additional arguments.
- **check_is_position_offset_only** – If `True` (the default), check that the `dr` and `drotation` values returned by *get_wcs_offset()* are sufficiently small to be considered a position offset only. If this is `False`, then the `dr` and `drotation` values are not checked.

Returns

`(dra_arcsec, ddec_arcsec)` tuple containing the offsets in arcseconds in the RA and Dec celestial coordinates between the current disc location (i.e. those returned by *BodyXY.get_disc_params()*) and the disc location calculated from the WCS information in the observation's FITS header.

Raises

`ValueError` – if no WCS information is found in the FITS header, or validation fails. A ValueError is also raised if `check_is_position_offset_only` is `True` and the `dr` or `drotation` values returned by *get_wcs_offset()* are not sufficiently small.

fit_disc_position() → None

Automatically find and set `x0` and `y0` so that the planet's disc is fit to the brightest part of the data.

fit_disc_radius() → None

Automatically find and set `r0` using aperture photometry.

This routine calculates the brightness in concentric annular apertures around `(x0, y0)` and sets `r0` as the radius where the brightness decreases the fastest. Note that this uses circular apertures, so will be less reliable for targets with greater flattening and may not work well for targets which are not entirely in the image frame.

Raises

`ValueError` – if `x0` or `y0` are not within the image frame.

get_mapped_data(*interpolation: Union[Literal['nearest', 'linear', 'quadratic', 'cubic'], int, tuple[int, int]] = 'linear'*, ***, *spline_smoothing: float = 0*, ***map_kwargs: Unpack[MapKwargs]*) → ndarray

Projects the observed *data* onto a map. See *BodyXY.generate_map_coordinates()* for details about customising the projection used.

For larger datasets, it can take some time to map every wavelength. Therefore, the mapped data is automatically cached (in a similar way to backplanes - see *BodyXY*) so that subsequent calls to this function do not have to recompute the mapped data. As with cached backplanes, the cached mapped data is automatically cleared if any disc parameters are changed (i.e. you shouldn't need to worry about the cache, it all happens 'magically' behind the scenes).

Parameters

- **interpolation** – Interpolation used when mapping. This can either any of `'nearest'`, `'linear'`, `'quadratic'` or `'cubic'`. Passed to *BodyXY.map_img()*.
- **spline_smoothing** – Passed to *BodyXY.map_img()*.
- ****map_kwargs** – Additional arguments are passed to *BodyXY.generate_map_coordinates()* to specify and customise the map projection.

Returns

Array containing cube of maped of the values in `img` at each location on the surface of the target body. Locations which are not visible or outside the projection domain have a value of NaN.

append_to_header(*keyword: str*, *value: str | float | bool | complex*, *comment: str | None = None*, *hierarch_keyword: bool = True*, *header: astropy.io.fits.header.Header | None = None*, *truncate_strings: bool = True*, *remove_existing: bool = True*)

Add a card to a FITS header. If a `header` is not specified, then *header* is modified.

By default, the keyword is modified to provide a consistent keyword prefix for all header cards added by this routine.

Parameters

- **keyword** – Card keyword.
- **value** – Card value.
- **comment** – Card comment. If unspecified not comment will be added.
- **hierarch_keyword** – Toggle adding the keyword prefix from *FITS_KEYWORD* to the keyword.
- **header** – FITS Header which the card will be added to in-place. If `header` is `None`, then *header* will be modified.
- **truncate_strings** – Allow string values to be truncated if they will create a card longer than 80 characters.
- **remove_existing** – Remove any existing cards with the same key before adding the new card.

add_header_metadata(*header: astropy.io.fits.header.Header | None = None*)

Add automatically generated metadata a FITS header. This is automatically called by `save()` so `add_header_metadata` does not normally need to be called manually.

Parameters

header – FITS Header which the metadata will be added to in-place. If `header` is `None`, then *header* will be modified.

make_filename(*extension: str = '.fits'*, *prefix: str = ''*, *suffix: str = ''*) → str

Automatically generates a useful filename from the target name and date of the observation, e.g. `'JUPITER_2000-01-01T123456.fits'`.

Parameters

- **extension** – Optionally specify the file extension to add to the filename.
- **prefix** – Optionally specify filename prefix.
- **suffix** – Optionally specify filename suffix.

Returns

Filename built from the target name and observation date.

save_observation(*path: str | os.PathLike*, ***, *include_wireframe: bool = True*, *wireframe_kwargs: dict[str, Any] | None = None*, *show_progress: bool = False*, *print_info: bool = True*) → None

Save a FITS file containing the observed data and generated backplanes.

The primary HDU in the FITS file will be the *data* and *header* of the observed data, with appropriate metadata automatically added to the header by *add_header_metadata()*. The backplanes are generated from all the registered backplanes in *BodyXY.backplanes* and are saved as additional HDUs in the FITS file.

For larger image sizes, the backplane generation can be slow, so this function may take some time to complete.

Parameters

- **path** – Filepath of output file.
- **include_wireframe** – Toggle generating and saving wireframe overlay image as an additional backplane of the output FITS file. The wireframe is generated by *BodyXY.get_wireframe_overlay_img()*.
- **wireframe_kwargs** – Dictionary of keyword arguments passed to *BodyXY.get_wireframe_overlay_img()* to customise the wireframe overlay.
- **show_progress** – Display a progress bar rather than printing progress info. This does not have an effect if `show_progress=True` was set when creating this `Observation`.
- **print_info** – Toggle printing of progress information (defaults to `True`).

save_mapped_observation(*path: str | os.PathLike*, ***, *interpolation: Union[Literal['nearest', 'linear', 'quadratic', 'cubic'], int, tuple[int, int]] = 'linear'*, *spline_smoothing: float = 0*, *include_backplanes: bool = True*, *include_wireframe: bool = True*, *wireframe_kwargs: dict[str, Any] | None = None*, *show_progress: bool = False*, *print_info: bool = True*, ***map_kwargs: Unpack[MapKwargs]*) → None

Save a FITS file containing the mapped observation in a cylindrical projection.

The mapped data is generated using *get_mapped_data()*, and mapped backplane data is saved by default.

For larger image sizes, the map projection and backplane generation can be slow, so this function may take some time to complete.

Parameters

- **path** – Filepath of output file.
- **interpolation** – Interpolation used when mapping. This can either any of `'nearest'`, `'linear'`, `'quadratic'` or `'cubic'`. Passed to *BodyXY.map_img()*.
- **spline_smoothing** – Passed to *BodyXY.map_img()*.

- **include_backplanes** – Toggle generating and saving backplanes to output FITS file.
- **include_wireframe** – Toggle generating and saving wireframe overlay map as an additional backplane of the output FITS file. The wireframe is generated by *BodyXY.get_wireframe_overlay_map()*.
- **wireframe_kwargs** – Dictionary of keyword arguments passed to *BodyXY.get_wireframe_overlay_map()* to customise the wireframe overlay.
- **show_progress** – Display a progress bar rather than printing progress info. This does not have an effect if `show_progress=True` was set when creating this `Observation`.
- **print_info** – Toggle printing of progress information (defaults to `True`).
- ****map_kwargs** – Additional arguments are passed to *BodyXY.generate_map_coordinates()* to specify and customise the map projection.

run_gui() → list[tuple[float, float]]

Run an interactive GUI to display and adjust the fitted observation.

This modifies the *Observation* object in-place, so can be used within a script to e.g. interactively fit the planet's disc. Simply run the GUI, adjust the parameters until the disc is fit, then close the GUI and the *Observation* object will have your new values:

```
# Load in some data
observation = planetmapper.Observation('exciting_data.fits')

# Use the GUI to manually fit the disc and set the x0,y0,r0,rotation values
observation.run_gui()

# At this point, you can use the manually fitted observation
observation.plot_wireframe_xy()
```

Hint: Once you have manually fitted the disc, you can simply close the user interface window and the disc parameters will be updated to the new values. This means that you don't need to click the *Save...* button unless you specifically want to save a navigated file to disk.

The return value can also be used to interactively select a locations::

```
observation = planetmapper.Observation('exciting_data.fits')
clicks = observation.run_gui()
ax = observation.plot_wireframe_radec()
for x, y in clicks:
    ra, dec = observation.xy2radec()
    ax.scatter(ra, dec)
```

See the *graphical user interface tutorial* for more details about the GUI.

Note: The *Open...* button is hidden for user interfaces created by this method to ensure that only one *Observation* object is modified by the user interface.

If you want the full user interface functionality instead, then call `planetmapper` from the command line or create and run a user interface manually using *planetmapper.gui.GUI.run()*.

Returns

List of `(x, y)` pixel coordinate tuples corresponding to where the user clicked on the plot window to mark a location.

class planetmapper.**BasicBody**(*target: str | int*, *utc: str | datetime.datetime | float | None = None*, *observer: str | int = 'EARTH'*, ***, *aberration_correction: str = 'CN'*, *observer_frame: str = 'J2000'*, ***kwargs*)

Bases: *BodyBase*

Class representing astronomical body which is treated as a point source.

This is typically used for objects which have limited data in the SPICE kernels (e.g. minor satellites which do not have well known radii). Usually, you are unlikely to need to create *BasicBody* instances directly, but they may be returned when using *Body.create_other_body()*.

This is a very simplified version of *Body*.

Parameters

- **target** – Name of target body (see *Body* for more details).
- **utc** – Time of observation (see *Body* for more details).
- **observer** – Name of observing body (see *Body* for more details).
- ****kwargs** – See *Body* for more details about additional arguments.

target: str

Name of the target body, as standardised by *SpiceBase.standardise_body_name()*.

utc: str

String representation of the time of the observation in the format `'2000-01-01T00:00:00.000000'`. This time is in the UTC timezone.

observer: str

Name of the observer body, as standardised by *SpiceBase.standardise_body_name()*.

aberration_correction: str

Aberration correction used to correct light travel time in SPICE.

observer_frame: str

Observer reference frame.

et: float

Ephemeris time of the observation corresponding to `utc`.

dtm: datetime.datetime

Python timezone aware datetime of the observation corresponding to `utc`.

target_body_id: int

SPICE numeric ID of the target body.

target_light_time: float

Light time from the target to the observer at the time of the observation.

target_distance: float

Distance from the target to the observer at the time of the observation.

target_ra: float

Right ascension (RA) of the target centre.

target_dec: float

Declination (Dec) of the target centre.

class planetmapper.**AngularCoordinateKwargs**

Bases: TypedDict

Class to help type hint keyword arguments of angular coordinate transformations and plotting functions.

See *Body.radec2angular()* for more details.

origin_ra: float | None

origin_dec: float | None

coordinate_rotation: float

class planetmapper.**WireframeKwargs**

Bases: TypedDict

Class to help type hint keyword arguments of wireframe plotting functions. The `color`, `alpha` and `zorder` parameters are a non-exhaustive list of commonly used formatting parameters which are passed to matplotlib functions.

See *Body.plot_wireframe_radec()* for more details.

label_poles: bool

add_title: bool

grid_interval: float

grid_lat_limit: float

indicate_equator: bool

indicate_prime_meridian: bool

formatting: dict[Literal['all', 'grid', 'equator', 'prime_meridian', 'limb', 'limb_illuminated', 'terminator', 'ring', 'pole', 'coordinate_of_interest_lonlat', 'coordinate_of_interest_radec', 'other_body_of_interest_marker', 'other_body_of_interest_label', 'hidden_other_body_of_interest_marker', 'hidden_other_body_of_interest_label', 'map_boundary'], dict[str, Any]] | None

color: str | tuple[float, float, float]

alpha: float

zorder: float

class planetmapper.**MapKwargs**

Bases: TypedDict

Class to help type hint keyword arguments of mapping functions.

See *BodyXY.generate_map_coordinates()* for more details.

projection: str

degree_interval: float

lon: float

lat: float

size: int

lon_coords: ndarray

lat_coords: ndarray

projection_x_coords: ndarray

projection_y_coords: numpy.ndarray | None

xlim: tuple[float, float] | None

ylim: tuple[float, float] | None

2.8 planetmapper.base

class planetmapper.base.**BodyBase**(*, *target: str | int*, *utc: str | datetime.datetime | float | None*, *observer: str | int*, *aberration_correction: str*, *observer_frame: str*, ***kwargs*)

Bases: *SpiceBase*

Base class for *planetmapper.Body* and *planetmapper.BasicBody*.

You are unlikely to need to use this class directly - use *planetmapper.Body* or *planetmapper.BasicBody* instead.

planetmapper.base.**load_kernels**(**paths: str*, *clear_before: bool = False*) → list[str]

Load spice kernels defined by patterns.

This function calls `spice.furnsh` on all kernels matching the provided patterns. The kernel paths returned by `glob.glob` are sorted by *sort_kernel_paths()* before being passed to `spice.furnsh`.

Hint: You generally don't need to call this function directly - it is called automatically the first time you create any object that inherits from *planetmapper.SpiceBase* (e.g. *planetmapper.Body* or *planetmapper.Observation*).

Parameters

- ***paths** – Paths to spice kernels, evaluated using `glob.glob` with `recursive=True`.
- **clear_before** – Clear kernel pool before loading new kernels.

planetmapper.base.**sort_kernel_paths**(*kernels: Collection[str]*) → list[str]

Sort kernel paths by path depth and alphabetically.

Kernels are sorted so that kernels in subdirectories are loaded before kernels in parent directories, and kernels in the same directory are sorted alphabetically. Kernels loaded later will take precedence over kernels loaded earlier, so this means that when kernels contain overlapping data:

- `spk/kernel.bsp` should take precedence over `spk/old/kernel.bsp`
- `kernel_101.bsp` should take precedence over `kernel_100.bsp`
- `a/kernel.bsp` should take precedence over `x/y/z/kernel.bsp`

> **Warning:** Although this function attempts to sort kernels in a sensible way, it is possible that it will not always do the right thing. If you have multiple kernels containing overlapping data (e.g. old predicted JWST ephemerides), it is generally safer to delete the old kernels, move them into a completely separate directory, or load them manually yourself using `spice.furnsh`.

Parameters
kernels – Collection of kernel paths.

Returns
Sorted list of kernel paths.

`planetmapper.base.prevent_kernel_loading() → None`

Prevent PlanetMapper from automatically loading kernels.

This function can be used if want to load kernels manually using `spice.furnsh`.

```
import spiceypy as spice
import planetmapper

# Call this function before creating any objects that inherit from SpiceBase,
# then load your desired kernels manually
planetmapper.base.prevent_kernel_loading()
kernels_to_load = [...]
for kernel in kernels_to_load:
    spice.furnsh(kernel)

# After setting up the kernels, you can use PlanetMapper as normal
body = planetmapper.Body('mars', '2021-01-01T00:00:00')
body.plot_wireframe_km()
```

Calling *clear_kernels()* will re-enable automatic kernel loading.

`planetmapper.base.clear_kernels() → None`

Clear spice kernel pool.

This function calls `spice.kclear()`, and also indicates to PlanetMapper that kernels will need to be reloaded when a new object is created.

`planetmapper.base.set_kernel_path(path: str | os.PathLike | None) → None`

Set the path of the directory containing SPICE kernels. See *the kernel directory documentation* for more detail.

Parameters
path – Directory which PlanetMapper will search for SPICE kernels. If `None`, then the default value of `'~/spice_kernels/'` will be used.

`planetmapper.base.get_kernel_path(return_source: Literal[False] = False) → str`
`planetmapper.base.get_kernel_path(return_source: Literal[True]) → tuple[str, str]`

Get the path of the directory of SPICE kernels used in PlanetMapper.

1. If a kernel path has been manually set using *set_kernel_path()*, then this path is used.
2. Otherwise the value of the environment variable `PLANETMAPPER_KERNEL_PATH` is used.
3. If `PLANETMAPPER_KERNEL_PATH` is not set, then the default value, `'~/spice_kernels/'` is used.

Parameters
return_source – If `True`, return a tuple of the kernel path and a string which indicates the

source of the kernel path. If `False` (the default), return only the kernel path. The possible source strings are: `'set_kernel_path()'`, `'PLANETMAPPER_KERNEL_PATH'`, and `'default'`.

Returns

The path of the directory of SPICE kernels used in PlanetMapper. If `return_source` is `True`, then a tuple of the kernel path and a string indicating the source of the kernel path is returned.

2.9 planetmapper.gui

Hint: See also the *page of examples* of using the PlanetMapper GUI

class planetmapper.gui.**GUI**(*allow_open: bool = True*)

Bases: `object`

Class to create and run graphical user interface to fit observations.

This class does not usually need to be run directly, as a GUI can be created directly from an *planetmapper.Observation* object using *planetmapper.Observation.run_gui()*, or by calling `planetmapper` from the command line.

click_locations: list[tuple[float, float]]

List of click locations marked on the plot in `(x, y)` pixel coordinates.

This list is cleared whenever a new observation is opened.

run() → None

Run the GUI.

set_observation(*observation:* Observation) → None

Set the observation used in the GUI.

For example, to run the GUI with the data in `'europa.fits'`, use:

```
gui = planetmapper.gui.GUI()
gui.set_observation(planetmapper.Observation('europa.fits'))
gui.run()
```

Parameters

observation – Observation to fit.

2.10 planetmapper.cli

This module is the entry point for the PlanetMapper Command Line Interface (CLI).

The command line interface is generally used to launch the PlanetMapper Graphical User Interface (GUI) without needing to write any Python code. For example, simply running `planetmapper` in the command line will launch the GUI.

For a list of available command line options, run `planetmapper --help` in the command line.

Hint: See also the *page of examples* of using the PlanetMapper GUI

2.11 planetmapper.utils

Various general helpful utilities.

planetmapper.utils.**format_radec_axes**(*ax: Axes*, *dec: float*, *dms_ticks: bool = True*, *add_axis_labels: bool = True*, *aspect_adjustable: Optional[Literal['box', 'datalim']] = 'datalim'*) → None

Format an axis to display RA/Dec coordinates nicely.

Parameters

- **ax** – Matplotlib axis to format.
- **dec** – Declination in degrees of centre of axis.
- **dms_ticks** – Toggle between showing ticks as degrees, minutes and seconds (e.g. 12°3456) or decimal degrees (e.g. 12.582).
- **add_axis_labels** – Add axis labels.
- **aspect_adjustable** – Set `adjustable` parameter when setting the aspect ratio. Passed to `matplotlib.axes.Axes.set_aspect()`. Set to None to skip setting the aspect ratio (generally this is only recommended if you're setting the aspect ratio yourself).

class planetmapper.utils.**DMSFormatter**

Bases: FuncFormatter

Matplotlib tick formatter to display angular values as degrees, minutes and seconds e.g. `12°3456`. Designed to work with *DMSLocator*.

```
ax = plt.cga()

ax.yaxis.set_major_locator(planetmapper.utils.DMSLocator())
ax.yaxis.set_major_formatter(planetmapper.utils.DMSFormatter())

ax.xaxis.set_major_locator(planetmapper.utils.DMSLocator())
ax.xaxis.set_major_formatter(planetmapper.utils.DMSFormatter())
```

class planetmapper.utils.**DMSLocator**

Bases: Locator

Matplotlib tick locator to display angular values as degrees, minutes and seconds. Designed to work with *DMSFormatter*.

```
ax = plt.cga()

ax.yaxis.set_major_locator(planetmapper.utils.DMSLocator())
ax.yaxis.set_major_formatter(planetmapper.utils.DMSFormatter())

ax.xaxis.set_major_locator(planetmapper.utils.DMSLocator())
ax.xaxis.set_major_formatter(planetmapper.utils.DMSFormatter())
```

planetmapper.utils.**decimal_degrees_to_dms**(*decimal_degrees: float*) → tuple[int, int, float]

Get degrees, minutes, seconds from decimal degrees.

`decimal_degrees_to_dms(-11.111)` returns `(-11.0, 6.0, 39.6)`.

Parameters

decimal_degrees – Decimal degrees.

Returns
`(degrees, minutes, seconds)` tuple

`planetmapper.utils.`**`decimal_degrees_to_dms_str`**(*decimal_degrees: float*, *seconds_fmt: str = ''*) → str

Create nicely formated DMS string from decimal degrees value (e.g. `'12°34'56"'`).

Uses *decimal_degrees_to_dms()* to perform the conversion.

Parameters

- **decimal_degrees** – Decimal degrees.
- **seconds_fmt** – Optionally specify a format string for the seconds part of the returned value. For example, `seconds_fmt='.3f'` will fix three decimal places for the fractional part of the seconds value.

Returns
String representting the degress, minutes, seconds of the angle.

class `planetmapper.utils.`**`ignore_warnings`**(**warining_strings: str*, ***kwargs*)

Bases: `catch_warnings`

Context manager to ignore general warnings using warnings.filterwarnings.

class `planetmapper.utils.`**`filter_fits_comment_warning`**(***, *record=False*, *module=None*, *action=None*, *category=<class 'Warning'>*, *lineno=0*, *append=False*)

Bases: `catch_warnings`

Context manager to hide FITS `Card is too long, comment will be truncated` warnings.

`planetmapper.utils.`**`normalise`**(*values: numpy.ndarray | list[float]*, *top: float = 1.0*, *bottom: float = 0.0*, *single_value: float | None = None*) → ndarray

Normalise iterable.

Parameters

- **values** – Iterable of values to normalise.
- **top** – Top of normalised range.
- **bottom** – Bottom of normalised range.
- **single_value** – If all values are the same, return this value.

Returns
Normalised values.

`planetmapper.utils.`**`check_path`**(*path: str*) → None

Checks if file path's directory tree exists, and creates it if necessary.

Assumes path is to a file if `os.path.split(path)[1]` contains '.', otherwise assumes path is to a directory.

2.12 planetmapper.data_loader

planetmapper.data_loader.**make_data_path**(*filename: str*) → str

Generates a path to a static data file.

Parameters
filename – Filename of the data file stored in `planetmapper/data`

Returns
Absolute path to the data file.

planetmapper.data_loader.**get_ring_radii**() → dict[str, dict[str, list[float]]]

Load planetary ring radii from data file.

These ring radii values are sourced from https://nssdc.gsfc.nasa.gov/planetary/planetfact.html.

Returns
Dictionary where the keys are planet names and the values are dictionaries containing ring data. These ring data dictionaries have keys corresponding to the names of the rings, and values with a list of ring radii in km. If the length of this list is 2, then the values give the inner and outer radii of the ring respectively. Otherwise, the length should be 1, meaning the ring has a single radius.

planetmapper.data_loader.**get_ring_aliases**() → dict[str, str]

Load ring aliases from data file.

These are used to allow pure ASCII ring names to be used in functions such as *planetmapper.Body.add_named_rings()*.

Returns
Dictionary where the keys are variants of ring names (e.g. `liberte`) and the values are the ring names (e.g. `liberté`) in a format consistent with the ring names in *get_ring_radii()*. Note that the keys and values are all in lower case.

2.13 planetmapper.kernel_downloader

Utility to help downloading spice kernels.

Will download local copy of kernels with same directory structure as on https://naif.jpl.nasa.gov/. Use *planetmapper.set_kernel_path()* to choose the location that the kernels are downloaded to.

These functions can be used to download a set of URLS. For example:

```
from planetmapper.kernel_downloader import download_urls

# Download all kernel files in generic_kernels/pck
download_urls('https://naif.jpl.nasa.gov/pub/naif/generic_kernels/pck/')

# Download specific kernel file
download_urls('https://naif.jpl.nasa.gov/pub/naif/generic_kernels/lsk/naif0012.tls')

# Download multiple sets of kernel files
download_urls(
    'https://naif.jpl.nasa.gov/pub/naif/generic_kernels/spk/planets/',
    'https://naif.jpl.nasa.gov/pub/naif/generic_kernels/spk/satellites/',
)
```

planetmapper.kernel_downloader.**download_urls**(**urls: str*, ***kwargs*) → None

Download data from naif.jpl.nasa.gov and save locally.

urls can either be a the url of a single kernel, or the index page containing multiple kernels.

If a single kernel, download the kernel using download_kernel().

If an index page, download all first-level files using *download_kernels_from_webpage()*.

Parameters

- **urls** – kernel URL on naif.jpl.nasa.gov.
- ****kwargs** – passed to *download_kernel()* and *download_kernels_from_webpage()*.

planetmapper.kernel_downloader.**download_kernels_from_webpage**(*index_url: str*, ***kwargs*) → None

Download all first-level kernels listed in the page given by index_url.

URL must be on https://naif.jpl.nasa.gov/pub/. This will break if JPL changes the format of the webpage.

Warning: This function will only download kernels found immediately on `index_url`. Kernels in nested folders must therefore be downloaded manually.

Parameters

- **index_url** – URL of index page on naif.jpl.nasa.gov.
- ****kwargs** – passed to *download_kernel()*.

planetmapper.kernel_downloader.**download_kernel**(*url: str*, *force_download: bool = False*, *note: str = ''*) → None

Download single kernel given by url.

URL must be on https://naif.jpl.nasa.gov/pub/. By default will only download file if if does not already exist locally. Set `force_download=True` to override this check and download the file even if it already exists locally.

Parameters

- **url** – URL of kernel on naif.jpl.nasa.gov.
- **force_download** – toggle overwriting already downloaded kernels.
- **note** – string to include in progress message.

planetmapper.kernel_downloader.**get_kernel_paths_from_webpage**(*index_url: str*) → list[str]

Get list of kernel urls from an index page on https://naif.jpl.nasa.gov/pub/.

This is a bit of a hack and will break if JPL changes the format of the webpage.

Parameters
index_url – URL of webpage.

Returns
List of URL strings corresponding to kernels on the webpage.

planetmapper.kernel_downloader.**download_file**(*url: str*, *local_path: str*) → None

Download kernel file to local system.

Parameters

- **url** – URL of kernel file.
- **local_path** – File path to save kernel file on local system.

2.14 Default backplanes

This page lists the backplanes which are automatically registered to every instance of *planetmapper.BodyXY*.

`LON-GRAPHIC` Planetographic longitude, positive W [deg]

- Image function: *planetmapper.BodyXY.get_lon_img()*
- Map function: *planetmapper.BodyXY.get_lon_map()*

`LAT-GRAPHIC` Planetographic latitude [deg]

- Image function: *planetmapper.BodyXY.get_lat_img()*
- Map function: *planetmapper.BodyXY.get_lat_map()*

`LON-CENTRIC` Planetocentric longitude [deg]

- Image function: *planetmapper.BodyXY.get_lon_centric_img()*
- Map function: *planetmapper.BodyXY.get_lon_centric_map()*

`LAT-CENTRIC` Planetocentric latitude [deg]

- Image function: *planetmapper.BodyXY.get_lat_centric_img()*
- Map function: *planetmapper.BodyXY.get_lat_centric_map()*

`RA` Right ascension [deg]

- Image function: *planetmapper.BodyXY.get_ra_img()*
- Map function: *planetmapper.BodyXY.get_ra_map()*

`DEC` Declination [deg]

- Image function: *planetmapper.BodyXY.get_dec_img()*
- Map function: *planetmapper.BodyXY.get_dec_map()*

`PIXEL-X` Observation x pixel coordinate [pixels]

- Image function: *planetmapper.BodyXY.get_x_img()*
- Map function: *planetmapper.BodyXY.get_x_map()*

`PIXEL-Y` Observation y pixel coordinate [pixels]

- Image function: *planetmapper.BodyXY.get_y_img()*
- Map function: *planetmapper.BodyXY.get_y_map()*

`KM-X` East-West distance in target plane [km]

- Image function: *planetmapper.BodyXY.get_km_x_img()*
- Map function: *planetmapper.BodyXY.get_km_x_map()*

`KM-Y` North-South distance in target plane [km]

- Image function: *planetmapper.BodyXY.get_km_y_img()*
- Map function: *planetmapper.BodyXY.get_km_y_map()*

`PHASE` Phase angle [deg]

- Image function: *planetmapper.BodyXY.get_phase_angle_img()*
- Map function: *planetmapper.BodyXY.get_phase_angle_map()*

`INCIDENCE` Incidence angle [deg]

- Image function: *planetmapper.BodyXY.get_incidence_angle_img()*
- Map function: *planetmapper.BodyXY.get_incidence_angle_map()*

`EMISSION` Emission angle [deg]

- Image function: *planetmapper.BodyXY.get_emission_angle_img()*
- Map function: *planetmapper.BodyXY.get_emission_angle_map()*

`AZIMUTH` Azimuth angle [deg]

- Image function: *planetmapper.BodyXY.get_azimuth_angle_img()*
- Map function: *planetmapper.BodyXY.get_azimuth_angle_map()*

`LOCAL-SOLAR-TIME` Local solar time [local hours]

- Image function: *planetmapper.BodyXY.get_local_solar_time_img()*
- Map function: *planetmapper.BodyXY.get_local_solar_time_map()*

`DISTANCE` Distance to observer [km]

- Image function: *planetmapper.BodyXY.get_distance_img()*
- Map function: *planetmapper.BodyXY.get_distance_map()*

`RADIAL-VELOCITY` Radial velocity away from observer [km/s]

- Image function: *planetmapper.BodyXY.get_radial_velocity_img()*

- Map function: *planetmapper.BodyXY.get_radial_velocity_map()*

`DOPPLER` Doppler factor, sqrt((1 + v/c)/(1 - v/c)) where v is radial velocity

- Image function: *planetmapper.BodyXY.get_doppler_img()*
- Map function: *planetmapper.BodyXY.get_doppler_map()*

`LIMB-DISTANCE` Distance above limb [km]

- Image function: *planetmapper.BodyXY.get_limb_distance_img()*
- Map function: *planetmapper.BodyXY.get_limb_distance_map()*

`LIMB-LON-GRAPHIC` Planetographic longitude of closest point on the limb [deg]

- Image function: *planetmapper.BodyXY.get_limb_lon_img()*
- Map function: *planetmapper.BodyXY.get_limb_lon_map()*

`LIMB-LAT-GRAPHIC` Planetographic latitude of closest point on the limb [deg]

- Image function: *planetmapper.BodyXY.get_limb_lat_img()*
- Map function: *planetmapper.BodyXY.get_limb_lat_map()*

`RING-RADIUS` Equatorial (ring) plane radius [km]

- Image function: *planetmapper.BodyXY.get_ring_plane_radius_img()*
- Map function: *planetmapper.BodyXY.get_ring_plane_radius_map()*

`RING-LON-GRAPHIC` Equatorial (ring) plane planetographic longitude [deg]

- Image function: *planetmapper.BodyXY.get_ring_plane_longitude_img()*
- Map function: *planetmapper.BodyXY.get_ring_plane_longitude_map()*

`RING-DISTANCE` Equatorial (ring) plane distance to observer [km]

- Image function: *planetmapper.BodyXY.get_ring_plane_distance_img()*
- Map function: *planetmapper.BodyXY.get_ring_plane_distance_map()*

2.14.1 Wireframe images

In addition to the above backplanes, a `WIREFRAME` backplane is also included by default in saved FITS files. This backplane contains a "wireframe" image of the body, which shows latitude/longitude gridlines, labels poles, displays the body's limb etc. These wireframe images can be used to help orient the observations, and can be used as an overlay if you are creating figures from the FITS files.

The wireframe images are a graphical guide rather than containing any scientific data, so they are not registered like the other backplanes. Note that the wireframe images have a fixed size, so they will not be the same size as the data/mapped data (although the aspect ratio will be the same).

- Image function: *planetmapper.BodyXY.get_wireframe_overlay_img()*
- Map function: *planetmapper.BodyXY.get_wireframe_overlay_map()*

2.15 Acknowledgements

PlanetMapper was developed by Oliver King at the University of Leicester with support from a European Research Council Consolidator Grant (under the European Union's Horizon 2020 research and innovation programme, grant agreement No 723890). Thanks to Mike Roman, Leigh Fletcher and Naomi Rowe-Gurney for their suggestions, beta testing and feedback!

2.16 Citation

2.16.1 JOSS Paper

If you use PlanetMapper in your research, please cite the following paper in the Journal of Open Source Software:

> King et al., (2023). PlanetMapper: A Python package for visualising, navigating and mapping Solar System observations. Journal of Open Source Software, 8(90), 5728, https://doi.org/10.21105/joss.05728

```
@article{king_2023_planetmapper,
  author  = {King, Oliver R. T. and Fletcher, Leigh N.},
  doi     = {10.21105/joss.05728},
  journal = {Journal of Open Source Software},
  month   = oct,
  number  = {90},
  pages   = {5728},
  title   = {{PlanetMapper: A Python package for visualising, navigating and mapping␣
↪Solar System observations}},
  url     = {https://joss.theoj.org/papers/10.21105/joss.05728},
  volume  = {8},
  year    = {2023}
}
```

2.16.2 Zenodo archive

Each individual version of PlanetMapper is archived on Zenodo at doi.org/10.5281/zenodo.7963121.

2.17 License

The PlanetMapper Python package is licensed under the MIT license:

2.18 Links

- GitHub repository
  - Issue tracker
  - Support discussions
  - Release notes (changelog)
  - Contributing guide
- Documentation
- PyPI page
- Conda Forge page
- Journal of Open Source Software paper
- *Help, support and contact details*

PYTHON MODULE INDEX

p

INDEX

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z