pfsspy Documentation

pfsspy contributors

Oct 22, 2020

CONTENTS

pfsspy is a python package for carrying out Potential Field Source Surface modelling, a commonly used magnetic field model of the Sun and other stars.

Note: Until pfsspy 1.0 is released, elements of the API are liable to change between versions. A full changelog that lists breaking changes, and how to adapt your code for them can be found at *Changelog*.

Note: If you find any bugs or have any suggestions for improvement, please raise an issue here: https://github.com/dstansby/pfsspy/issues

---

pfsspy can be installed from PyPi using

```
pip install pfsspy
```

This will install pfsspy and all of its dependencies. In addition to the core dependencies, there are two optional dependencies (numba, streamtracer) that improve code performance. These can be installed with

```
pip install pfsspy[performance]
```

CHAPTER

ONE

CITING

If you use pfsspy in work that results in publication, please cite the archived code at *both*

- https://zenodo.org/record/2566462
- https://zenodo.org/record/1472183

Citation details can be found at the lower right hand of each web page.

CHAPTER

TWO

CONTENTS

2.1 Examples

2.1.1 Using pfsspy

Magnetic field along a field line

How to get the value of the magnetic field along a field line traced through the PFSS solution.

First, import required modules

```
import astropy.constants as const
import astropy.units as u
from astropy.coordinates import SkyCoord
import matplotlib.pyplot as plt
import numpy as np
import sunpy.map

import pfsspy
from pfsspy import tracing
from pfsspy.sample_data import get_gong_map
```

Load a GONG magnetic field map. If 'gong.fits' is present in the current directory, just use that, otherwise download a sample GONG map.

```
gong_fname = get_gong_map()
```

Out:

```
Files Downloaded:   0%|          | 0/1 [00:00<?, ?file/s]

mrzqs200901t1304c2234_022.fits.gz:   0%|          | 0.00/242k [00:00<?, ?B/s][A

mrzqs200901t1304c2234_022.fits.gz:   0%|          | 100/242k [00:00<07:48, 515B/s][A

mrzqs200901t1304c2234_022.fits.gz:  87%|########6 | 209k/242k [00:00<00:43, 735B/s][A
Files Downloaded: 100%|##########| 1/1 [00:00<00:00,  1.80file/s]
Files Downloaded: 100%|##########| 1/1 [00:00<00:00,  1.80file/s]


                                                                                [A
```

We can now use SunPy to load the GONG fits file, and extract the magnetic field data.

The mean is subtracted to remove the monopole component

```
gong_map = sunpy.map.Map(gong_fname)
# Remove the mean
gong_map = sunpy.map.Map(gong_map.data - np.mean(gong_map.data), gong_map.meta)
```

The PFSS solution is calculated on a regular 3D grid in (phi, s, rho), where rho = ln(r), and r is the standard spherical radial coordinate. We need to define the number of rho grid points, and the source surface radius.

```
nrho = 35
rss = 2.5
```

From the boundary condition, number of radial grid points, and source surface, we now construct an Input object that stores this information

```
input = pfsspy.Input(gong_map, nrho, rss)
output = pfsspy.pfss(input)
```

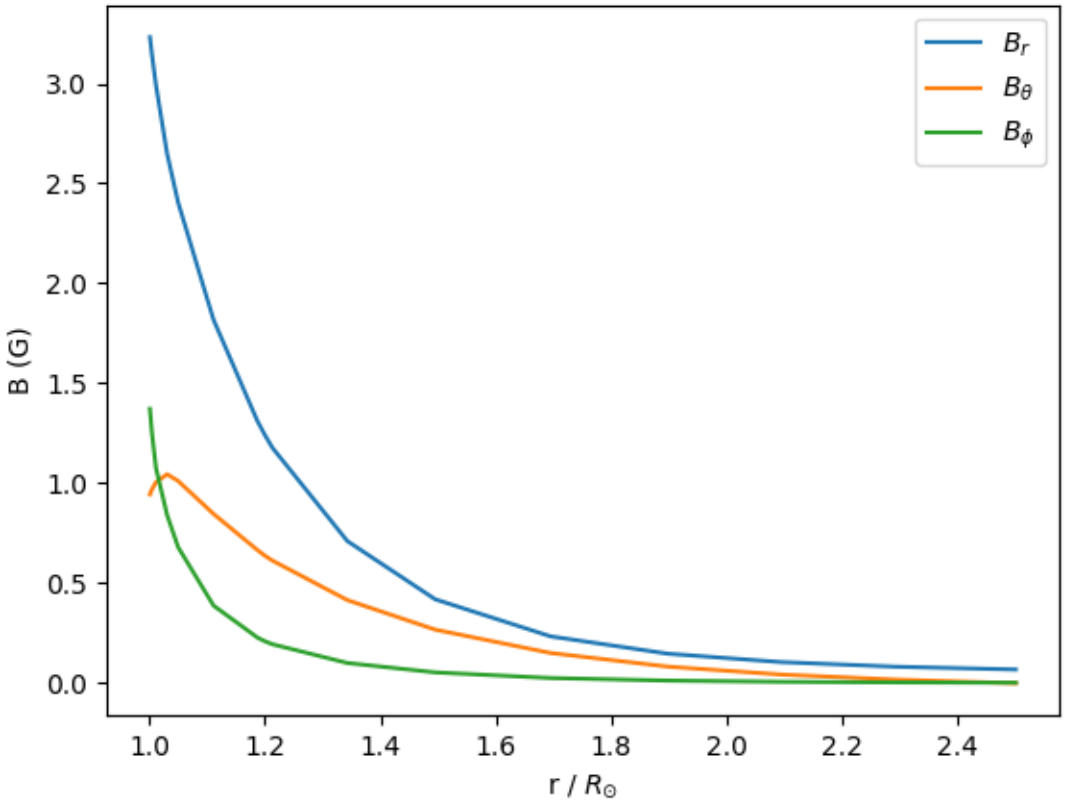
Now take a seed point, and trace a magnetic field line through the PFSS solution from this point

```
tracer = tracing.PythonTracer()
r = 1.2 * const.R_sun
lat = 70 * u.deg
lon = 0 * u.deg

seeds = SkyCoord(lon, lat, r, frame=output.coordinate_frame)
field_lines = tracer.trace(seeds, output)
```

From this field line we can extract the coordinates, and then use `.b_along_fline` to get the components of the magnetic field along the field line.

From the plot we can see that the non-radial component of the mangetic field goes to zero at the source surface, as expected.

```
field_line = field_lines[0]
B = field_line.b_along_fline
r = field_line.coords.radius
fig, ax = plt.subplots()

ax.plot(r.to(const.R_sun), B[:, 0], label=r'$B_{r}$')
ax.plot(r.to(const.R_sun), B[:, 1], label=r'$B_{\theta}$')
ax.plot(r.to(const.R_sun), B[:, 2], label=r'$B_{\phi}$')
ax.legend()
ax.set_xlabel(r'r / $R_{\odot}$')
ax.set_ylabel(f'B ({B.unit})')

plt.show()
```



Total running time of the script: (0 minutes 7.430 seconds)

Open/closed field map

Creating an open/closed field map on the solar surface.

First, import required modules

```
import astropy.units as u
import astropy.constants as const
from astropy.coordinates import SkyCoord
import matplotlib.pyplot as plt
import matplotlib.colors as mcolor

import numpy as np
import sunpy.map

import pfsspy
from pfsspy import tracing
from pfsspy.sample_data import get_gong_map
```

Load a GONG magnetic field map. If 'gong.fits' is present in the current directory, just use that, otherwise download a sample GONG map.

```
gong_fname = get_gong_map()
```

We can now use SunPy to load the GONG fits file, and extract the magnetic field data.

The mean is subtracted to enforce div(B) = 0 on the solar surface: n.b. it is not obvious this is the correct way to do this, so use the following lines at your own risk!

```
gong_map = sunpy.map.Map(gong_fname)
# Remove the mean
gong_map = sunpy.map.Map(gong_map.data - np.mean(gong_map.data), gong_map.meta)
```

Set the model parameters

```
nrho = 60
rss = 2.5
```

Construct the input, and calculate the output solution

```
input = pfsspy.Input(gong_map, nrho, rss)
output = pfsspy.pfss(input)
```

Finally, using the 3D magnetic field solution we can trace some field lines. In this case a grid of 90 x 180 points equally gridded in theta and phi are chosen and traced from the source surface outwards.

First, set up the tracing seeds

```
r = const.R_sun
# Number of steps in cos(latitude)
nsteps = 90
lon_1d = np.linspace(0, 2 * np.pi, nsteps * 2 + 1)
lat_1d = np.arcsin(np.linspace(-1, 1, nsteps + 1))
lon, lat = np.meshgrid(lon_1d, lat_1d, indexing='ij')
lon, lat = lon*u.rad, lat*u.rad
seeds = SkyCoord(lon.ravel(), lat.ravel(), r, frame=output.coordinate_frame)
```

Trace the field lines

```
print('Tracing field lines...')
tracer = tracing.FortranTracer(max_steps=2000)
field_lines = tracer.trace(seeds, output)
print('Finished tracing field lines')
```

Plot the result. The to plot is the input magnetogram, and the bottom plot shows a contour map of the the footpoint polarities, which are +/- 1 for open field regions and 0 for closed field regions.

```
fig = plt.figure()
m = input.map
ax = fig.add_subplot(2, 1, 1, projection=m)
m.plot()
ax.set_title('Input GONG magnetogram')

ax = fig.add_subplot(2, 1, 2)
cmap = mcolor.ListedColormap(['tab:red', 'black', 'tab:blue'])
norm = mcolor.BoundaryNorm([-1.5, -0.5, 0.5, 1.5], ncolors=3)
pols = field_lines.polarities.reshape(2 * nsteps + 1, nsteps + 1).T
ax.contourf(np.rad2deg(lon_1d), np.sin(lat_1d), pols, norm=norm, cmap=cmap)
ax.set_ylabel('sin(latitude)')

ax.set_title('Open (blue/red) and closed (black) field')
ax.set_aspect(0.5 * 360 / 2)
```

(continues on next page)

(continued from previous page)

```
plt.show()
```

Total running time of the script: (0 minutes 0.000 seconds)

Dipole source solution

A simple example showing how to use pfsspy to compute the solution to a dipole source field.

First, import required modules

```
import astropy.constants as const
import astropy.units as u
from astropy.coordinates import SkyCoord
from astropy.time import Time
import matplotlib.pyplot as plt
import matplotlib.patches as mpatch
import numpy as np
import sunpy.map
import pfsspy
import pfsspy.coords as coords
```

To start with we need to construct an input for the PFSS model. To do this, first set up a regular 2D grid in (phi, s), where s = cos(theta) and (phi, theta) are the standard spherical coordinate system angular coordinates. In this case the resolution is (360 x 180).

```
nphi = 360
ns = 180

phi = np.linspace(0, 2 * np.pi, nphi)
s = np.linspace(-1, 1, ns)
s, phi = np.meshgrid(s, phi)
```

Now we can take the grid and calculate the boundary condition magnetic field.

```
def dipole_Br(r, s):
    return 2 * s / r**3


br = dipole_Br(1, s)
```

The PFSS solution is calculated on a regular 3D grid in (phi, s, rho), where rho = ln(r), and r is the standard spherical radial coordinate. We need to define the number of rho grid points, and the source surface radius.

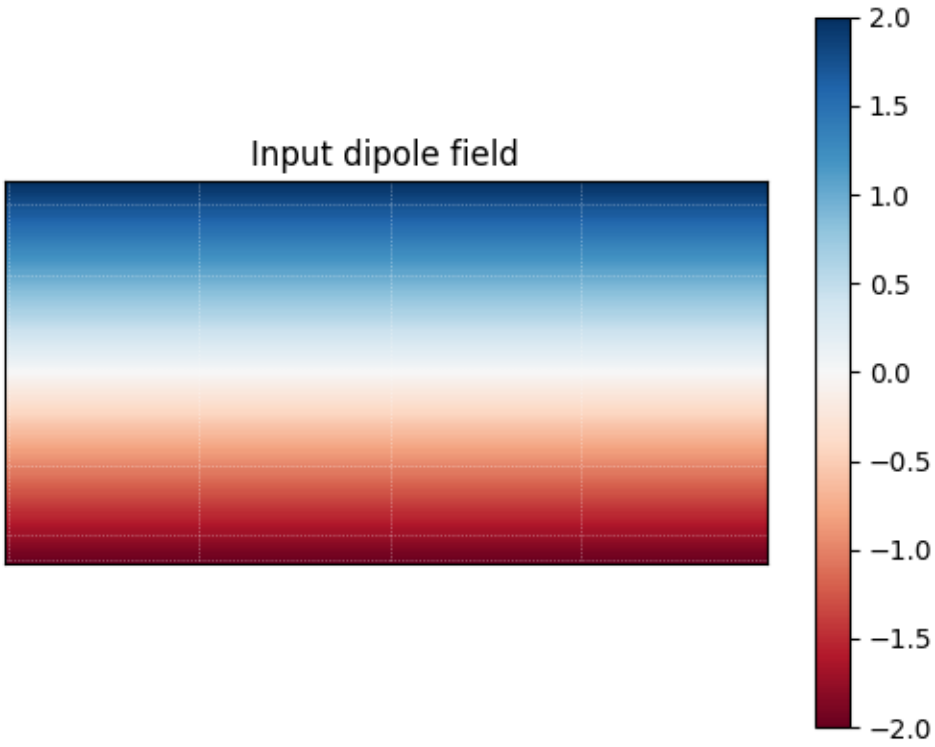
```
nrho = 30
rss = 2.5
```

From the boundary condition, number of radial grid points, and source surface, we now construct an Input object that stores this information

```
header = pfsspy.utils.carr_cea_wcs_header(Time('2020-1-1'), br.shape)
input_map = sunpy.map.Map((br.T, header))
input = pfsspy.Input(input_map, nrho, rss)
```

Using the Input object, plot the input field

```
m = input.map
fig = plt.figure()
ax = plt.subplot(projection=m)
m.plot()
plt.colorbar()
ax.set_title('Input dipole field')
```



Out:

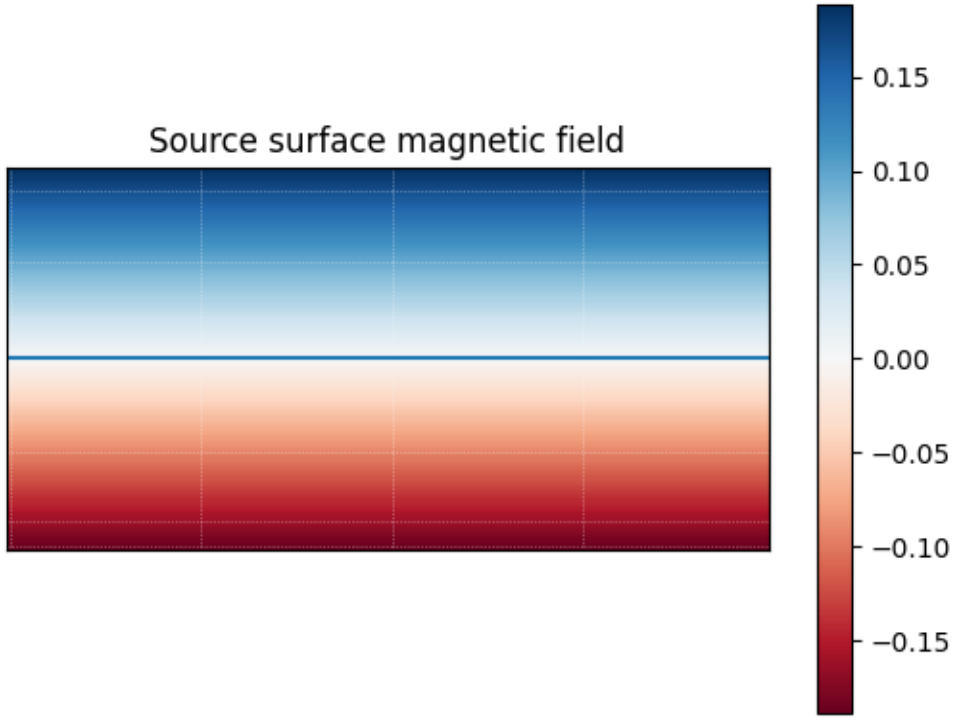
```
/home/docs/checkouts/readthedocs.org/user_builds/pfsspy/envs/0.6.3/lib/python3.7/site-
↪packages/astropy/visualization/wcsaxes/core.py:211: MatplotlibDeprecationWarning:␣
↪Passing parameters norm and vmin/vmax simultaneously is deprecated since 3.3 and␣
↪will become an error two minor releases later. Please pass vmin/vmax directly to␣
↪the norm when creating it.
  return super().imshow(X, *args, origin=origin, **kwargs)

Text(0.5, 1.0, 'Input dipole field')
```

Now calculate the PFSS solution.

```
output = pfsspy.pfss(input)
```

Using the Output object we can plot the source surface field, and the polarity inversion line.

```
ss_br = output.source_surface_br

# Create the figure and axes
fig = plt.figure()
ax = plt.subplot(projection=ss_br)

# Plot the source surface map
ss_br.plot()
# Plot the polarity inversion line
ax.plot_coord(output.source_surface_pils[0])
# Plot formatting
plt.colorbar()
ax.set_title('Source surface magnetic field')
```



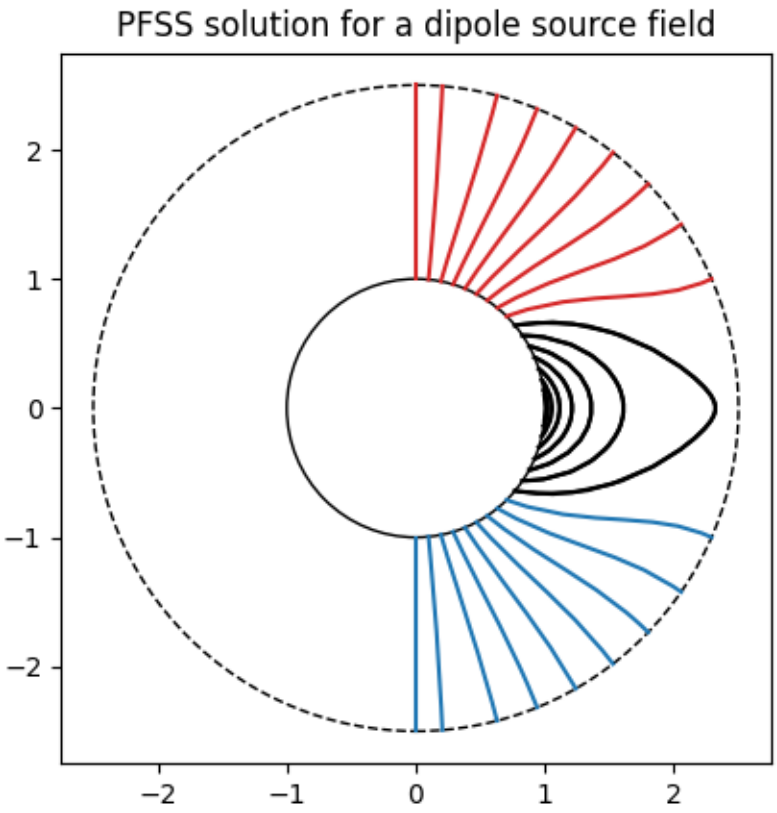
Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/pfsspy/envs/0.6.3/lib/python3.7/site-
↪packages/astropy/visualization/wcsaxes/core.py:211: MatplotlibDeprecationWarning:␣
↪Passing parameters norm and vmin/vmax simultaneously is deprecated since 3.3 and␣
↪will become an error two minor releases later. Please pass vmin/vmax directly to␣
↪the norm when creating it.
  return super().imshow(X, *args, origin=origin, **kwargs)

Text(0.5, 1.0, 'Source surface magnetic field')
```

Finally, using the 3D magnetic field solution we can trace some field lines. In this case 32 points equally spaced in

theta are chosen and traced from the source surface outwards.

```
fig, ax = plt.subplots()
ax.set_aspect('equal')

# Take 32 start points spaced equally in theta
r = 1.01 * const.R_sun
lon = np.pi / 2 * u.rad
lat = np.linspace(-np.pi / 2, np.pi / 2, 33) * u.rad
seeds = SkyCoord(lon, lat, r, frame=output.coordinate_frame)

tracer = pfsspy.tracing.PythonTracer()
field_lines = tracer.trace(seeds, output)

for field_line in field_lines:
    coords = field_line.coords
    coords.representation_type = 'cartesian'
    color = {0: 'black', -1: 'tab:blue', 1: 'tab:red'}.get(field_line.polarity)
    ax.plot(coords.y / const.R_sun,
            coords.z / const.R_sun, color=color)

# Add inner and outer boundary circles
ax.add_patch(mpatch.Circle((0, 0), 1, color='k', fill=False))
ax.add_patch(mpatch.Circle((0, 0), input.grid.rss, color='k', linestyle='--',
                           fill=False))
ax.set_title('PFSS solution for a dipole source field')
plt.show()
```



Total running time of the script: (0 minutes 5.685 seconds)

GONG PFSS extrapolation

Calculating PFSS solution for a GONG synoptic magnetic field map.

First, import required modules

```
import astropy.constants as const
import astropy.units as u
from astropy.coordinates import SkyCoord
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
import sunpy.map

import pfsspy
from pfsspy import coords
from pfsspy import tracing
from pfsspy.sample_data import get_gong_map
```

Load a GONG magnetic field map. If ‘gong.fits’ is present in the current directory, just use that, otherwise download a sample GONG map.

```
gong_fname = get_gong_map()
```

We can now use SunPy to load the GONG fits file, and extract the magnetic field data.

The mean is subtracted to enforce div(B) = 0 on the solar surface: n.b. it is not obvious this is the correct way to do this, so use the following lines at your own risk!

```
gong_map = sunpy.map.Map(gong_fname)
# Remove the mean
gong_map = sunpy.map.Map(gong_map.data - np.mean(gong_map.data), gong_map.meta)
```

The PFSS solution is calculated on a regular 3D grid in (phi, s, rho), where rho = ln(r), and r is the standard spherical radial coordinate. We need to define the number of rho grid points, and the source surface radius.

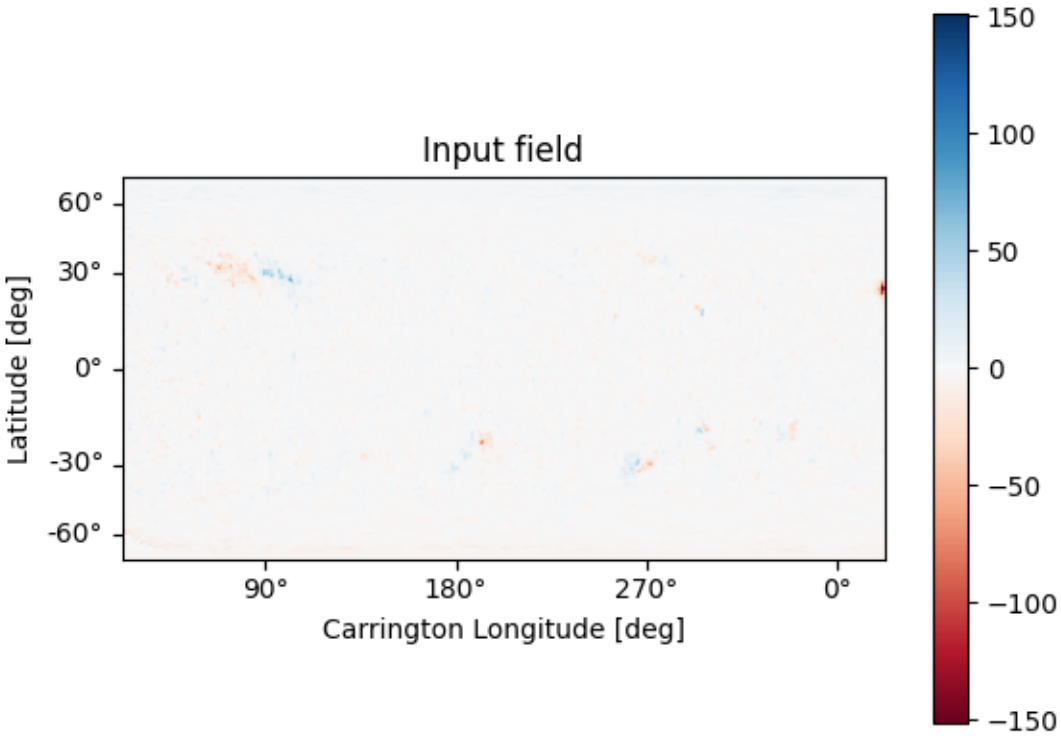
```
nrho = 35
rss = 2.5
```

From the boundary condition, number of radial grid points, and source surface, we now construct an Input object that stores this information

```
input = pfsspy.Input(gong_map, nrho, rss)


def set_axes_lims(ax):
    ax.set_xlim(0, 360)
    ax.set_ylim(0, 180)
```

Using the Input object, plot the input field

```
m = input.map
fig = plt.figure()
ax = plt.subplot(projection=m)
m.plot()
plt.colorbar()
ax.set_title('Input field')
set_axes_lims(ax)
```



Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/pfsspy/envs/0.6.3/lib/python3.7/site-
↪packages/astropy/visualization/wcsaxes/core.py:211: MatplotlibDeprecationWarning:␣
↪Passing parameters norm and vmin/vmax simultaneously is deprecated since 3.3 and␣
↪will become an error two minor releases later. Please pass vmin/vmax directly to␣
↪the norm when creating it.
  return super().imshow(X, *args, origin=origin, **kwargs)
```

Now calculate the PFSS solution, and plot the polarity inversion line.

```
output = pfsspy.pfss(input)
# output.plot_pil(ax)
```

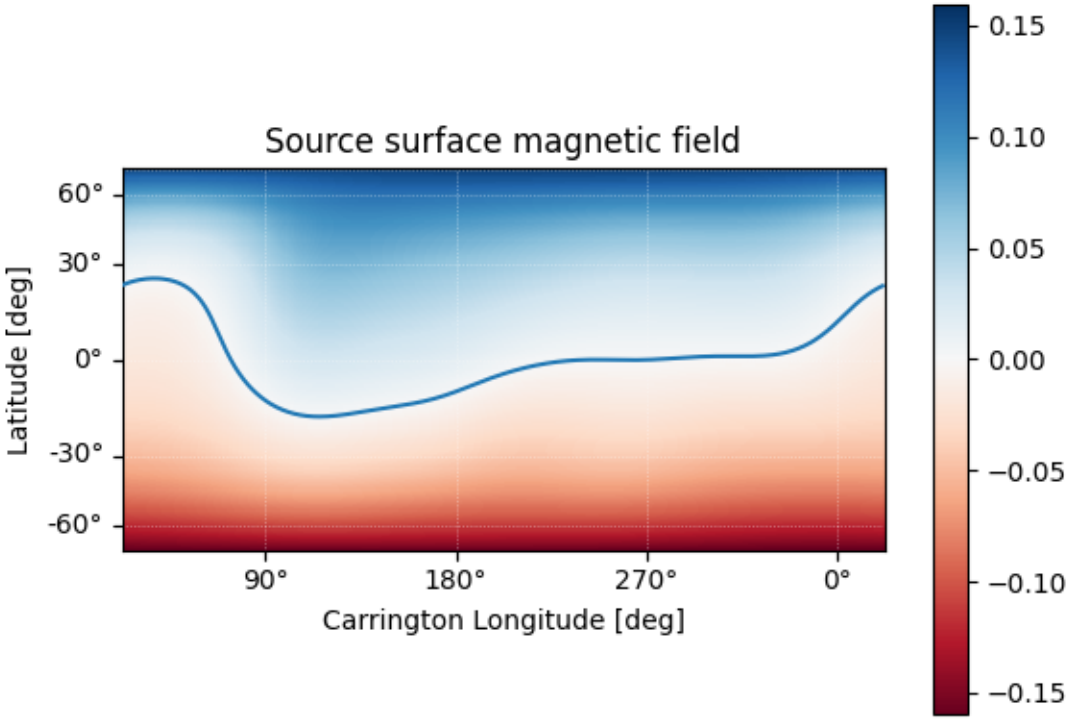
Using the Output object we can plot the source surface field, and the polarity inversion line.

```
ss_br = output.source_surface_br
# Create the figure and axes
fig = plt.figure()
ax = plt.subplot(projection=ss_br)

# Plot the source surface map
ss_br.plot()
# Plot the polarity inversion line
ax.plot_coord(output.source_surface_pils[0])
# Plot formatting
```

(continues on next page)

(continued from previous page)

```
plt.colorbar()
ax.set_title('Source surface magnetic field')
set_axes_lims(ax)
```



Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/pfsspy/envs/0.6.3/lib/python3.7/site-
↪packages/astropy/visualization/wcsaxes/core.py:211: MatplotlibDeprecationWarning:␣
↪Passing parameters norm and vmin/vmax simultaneously is deprecated since 3.3 and␣
↪will become an error two minor releases later. Please pass vmin/vmax directly to␣
↪the norm when creating it.
  return super().imshow(X, *args, origin=origin, **kwargs)
```

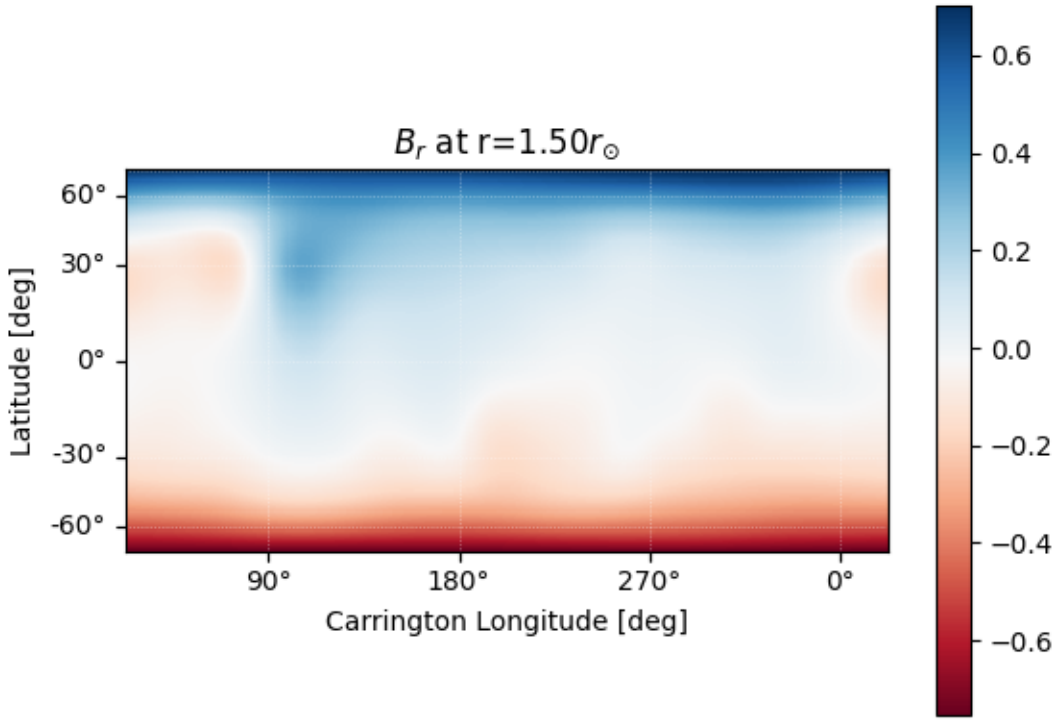
It is also easy to plot the magnetic field at an arbitrary height within the PFSS solution.

```
# Get the radial magnetic field at a given height
ridx = 15
br = output.bc[0][:, :, ridx]
# Create a sunpy Map object using output WCS
br = sunpy.map.Map(br.T, output.source_surface_br.wcs)
# Get the radial coordinate
r = np.exp(output.grid.rc[ridx])

# Create the figure and axes
```

(continues on next page)

(continued from previous page)

```
fig = plt.figure()
ax = plt.subplot(projection=br)

# Plot the source surface map
br.plot(cmap='RdBu')
# Plot formatting
plt.colorbar()
ax.set_title('$B_{r}$ ' + f'at r={r:.2f}' + '$r_{\\odot}$')
set_axes_lims(ax)
```



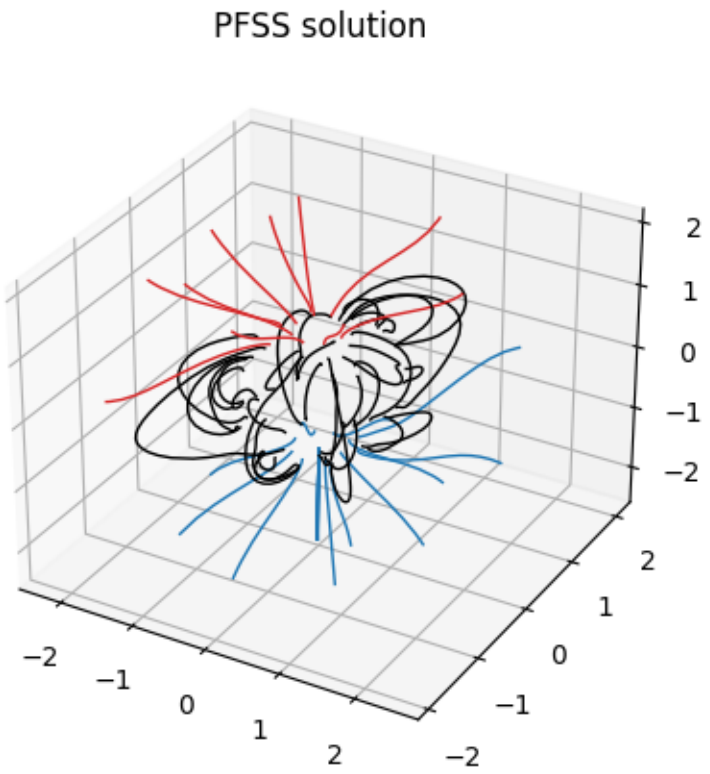
Finally, using the 3D magnetic field solution we can trace some field lines. In this case 64 points equally gridded in theta and phi are chosen and traced from the source surface outwards.

```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

tracer = tracing.PythonTracer()
r = 1.2 * const.R_sun
lat = np.linspace(-np.pi / 2, np.pi / 2, 8, endpoint=False)
lon = np.linspace(0, 2 * np.pi, 8, endpoint=False)
lat, lon = np.meshgrid(lat, lon, indexing='ij')
lat, lon = lat.ravel() * u.rad, lon.ravel() * u.rad

seeds = SkyCoord(lon, lat, r, frame=output.coordinate_frame)
```

(continues on next page)

(continued from previous page)

```
field_lines = tracer.trace(seeds, output)

for field_line in field_lines:
    color = {0: 'black', -1: 'tab:blue', 1: 'tab:red'}.get(field_line.polarity)
    coords = field_line.coords
    coords.representation_type = 'cartesian'
    ax.plot(coords.x / const.R_sun,
            coords.y / const.R_sun,
            coords.z / const.R_sun,
            color=color, linewidth=1)

ax.set_title('PFSS solution')
plt.show()

# sphinx_gallery_thumbnail_number = 4
```



Total running time of the script: (0 minutes 9.194 seconds)

Overplotting field lines on AIA maps

This example shows how to take a PFSS solution, trace some field lines, and overplot the traced field lines on an AIA 193 map.

First, we import the required modules

```
from datetime import datetime
import os

import astropy.constants as const
import astropy.units as u
from astropy.coordinates import SkyCoord
import matplotlib.pyplot as plt
import numpy as np
import sunpy.map
import sunpy.io.fits

import pfsspy
import pfsspy.tracing as tracing
from pfsspy.sample_data import get_gong_map
```

Load a GONG magnetic field map. If 'gong.fits' is present in the current directory, just use that, otherwise download a sample GONG map.

```
gong_fname = get_gong_map()
```

We can now use SunPy to load the GONG fits file, and extract the magnetic field data.

The mean is subtracted to enforce div(B) = 0 on the solar surface: n.b. it is not obvious this is the correct way to do this, so use the following lines at your own risk!

```
gong_map = sunpy.map.Map(gong_fname)
# Remove the mean
gong_map = sunpy.map.Map(gong_map.data - np.mean(gong_map.data), gong_map.meta)
```

Load the corresponding AIA 193 map

```
if not os.path.exists('aia_map.fits'):
    import urllib.request
    urllib.request.urlretrieve(
        'http://jsoc2.stanford.edu/data/aia/synoptic/2020/09/01/H1300/AIA20200901_
↪1300_0193.fits',
        'aia_map.fits')

aia = sunpy.map.Map('aia_map.fits')
dtime = aia.date
```

The PFSS solution is calculated on a regular 3D grid in (phi, s, rho), where rho = ln(r), and r is the standard spherical radial coordinate. We need to define the number of grid points in rho, and the source surface radius.

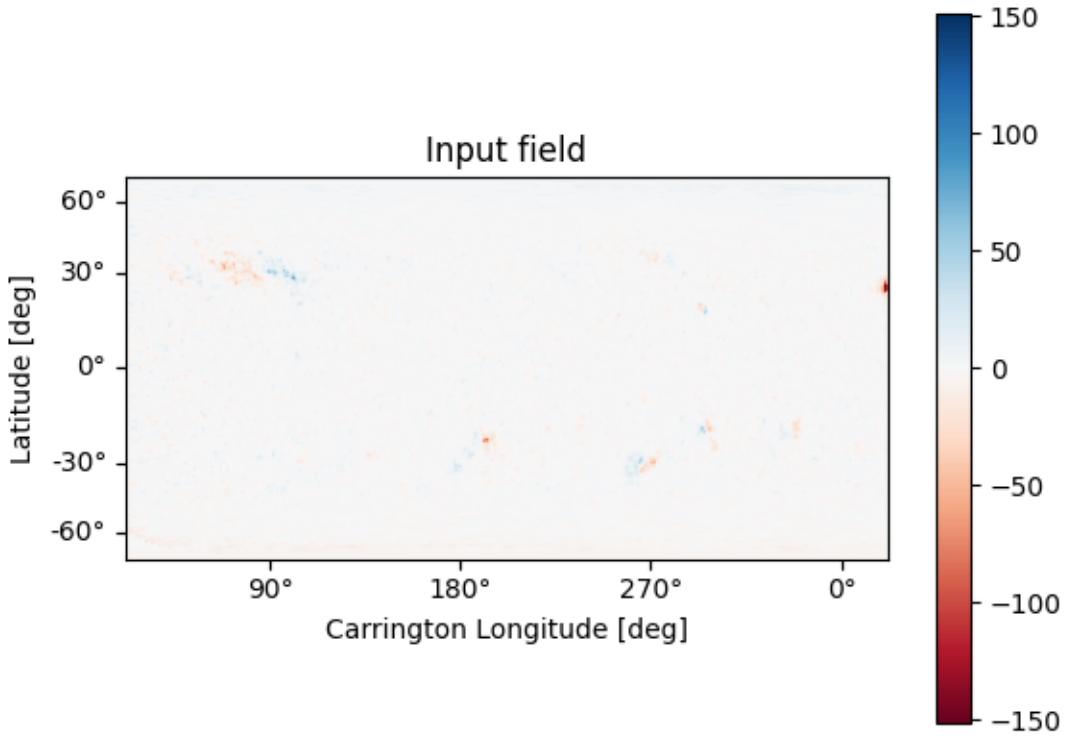
```
nrho = 25
rss = 2.5
```

From the boundary condition, number of radial grid points, and source surface, we now construct an `Input` object that stores this information

```
input = pfsspy.Input(gong_map, nrho, rss)
```

Using the `Input` object, plot the input photospheric magnetic field

```
m = input.map
fig = plt.figure()
ax = plt.subplot(projection=m)
m.plot()
plt.colorbar()
ax.set_title('Input field')
```

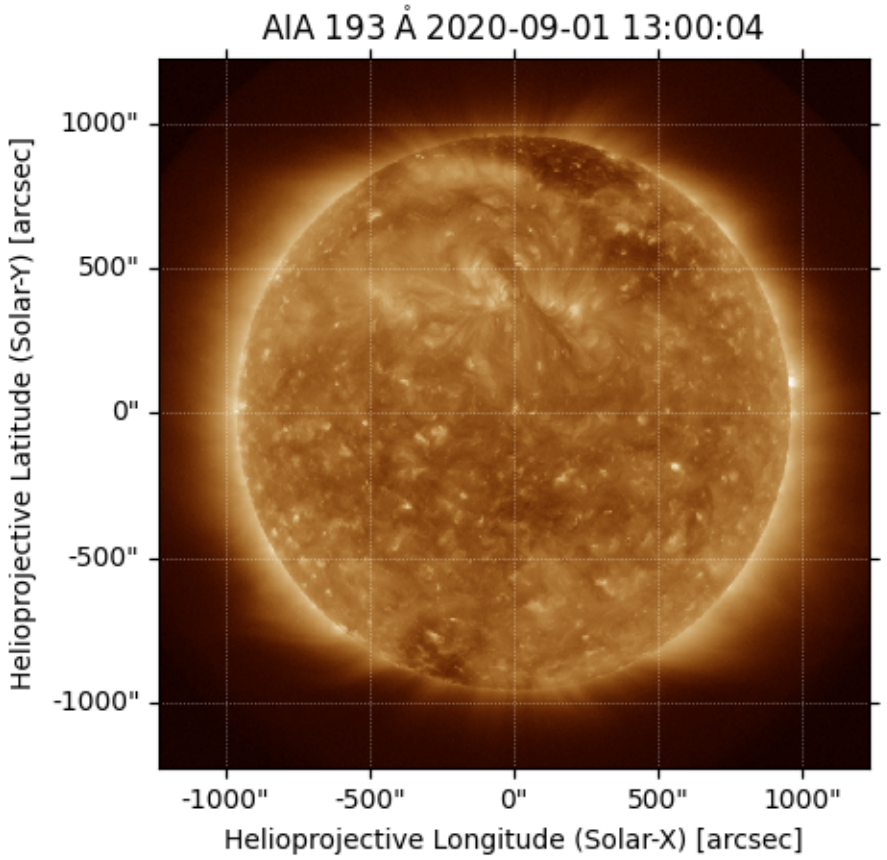


Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/pfsspy/envs/0.6.3/lib/python3.7/site-
↪packages/astropy/visualization/wcsaxes/core.py:211: MatplotlibDeprecationWarning:␣
↪Passing parameters norm and vmin/vmax simultaneously is deprecated since 3.3 and␣
↪will become an error two minor releases later. Please pass vmin/vmax directly to␣
↪the norm when creating it.
  return super().imshow(X, *args, origin=origin, **kwargs)

Text(0.5, 1.0, 'Input field')
```

We can also plot the AIA map to give an idea of the global picture. There is a nice active region in the top right of the AIA plot, that can also be seen in the top left of the photospheric field plot above.

```
ax = plt.subplot(1, 1, 1, projection=aia)
aia.plot(ax)
```



Out:

```
<matplotlib.image.AxesImage object at 0x7febf2570d90>
```

Now we construct a 5 x 5 grid of footpoitns to trace some magnetic field lines from. These coordinates are defined in the native Carrington coordinates of the input magnetogram.

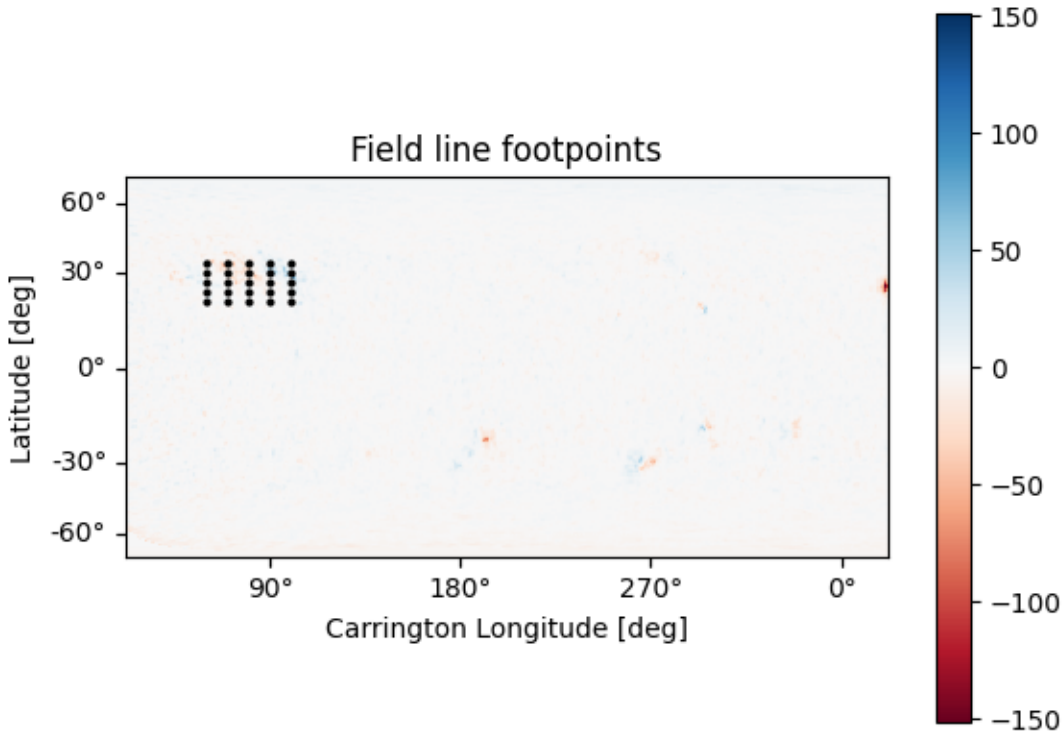
```
# Create 5 points spaced between sin(lat)={0.35, 0.55}
s = np.linspace(0.35, 0.55, 5)
# Create 5 points spaced between long={60, 100} degrees
phi = np.linspace(60, 100, 5)
print(f's = {s}')
print(f'phi = {phi}')
# Make a 2D grid from these 1D points
s, phi = np.meshgrid(s, phi)

# Now convert the points to a coordinate object
lat = np.arcsin(s) * u.rad
lon = phi * u.deg
seeds = SkyCoord(lon.ravel(), lat.ravel(), 1.01 * const.R_sun,
                 frame=gong_map.coordinate_frame)
```

Out:

```
s = [0.35 0.4  0.45 0.5  0.55]
phi = [ 60.  70.  80.  90. 100.]
```

Plot the magnetogram and the seed footpoints The footpoints are centered around the active region metnioned above.

```
m = input.map
fig = plt.figure()
ax = plt.subplot(projection=m)
m.plot()
plt.colorbar()

ax.plot_coord(seeds, color='black', marker='o', linewidth=0, markersize=2)

# Set the axes limits. These limits have to be in pixel values
# ax.set_xlim(0, 180)
# ax.set_ylim(45, 135)
ax.set_title('Field line footpoints')
ax.set_ylim(bottom=0)
```



Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/pfsspy/envs/0.6.3/lib/python3.7/site-
↪packages/astropy/visualization/wcsaxes/core.py:211: MatplotlibDeprecationWarning:␣
↪Passing parameters norm and vmin/vmax simultaneously is deprecated since 3.3 and␣
↪will become an error two minor releases later. Please pass vmin/vmax directly to␣
↪the norm when creating it.
```

(continues on next page)

(continued from previous page)

```
  return super().imshow(X, *args, origin=origin, **kwargs)

(0.0, 179.5)
```

Compute the PFSS solution from the GONG magnetic field input

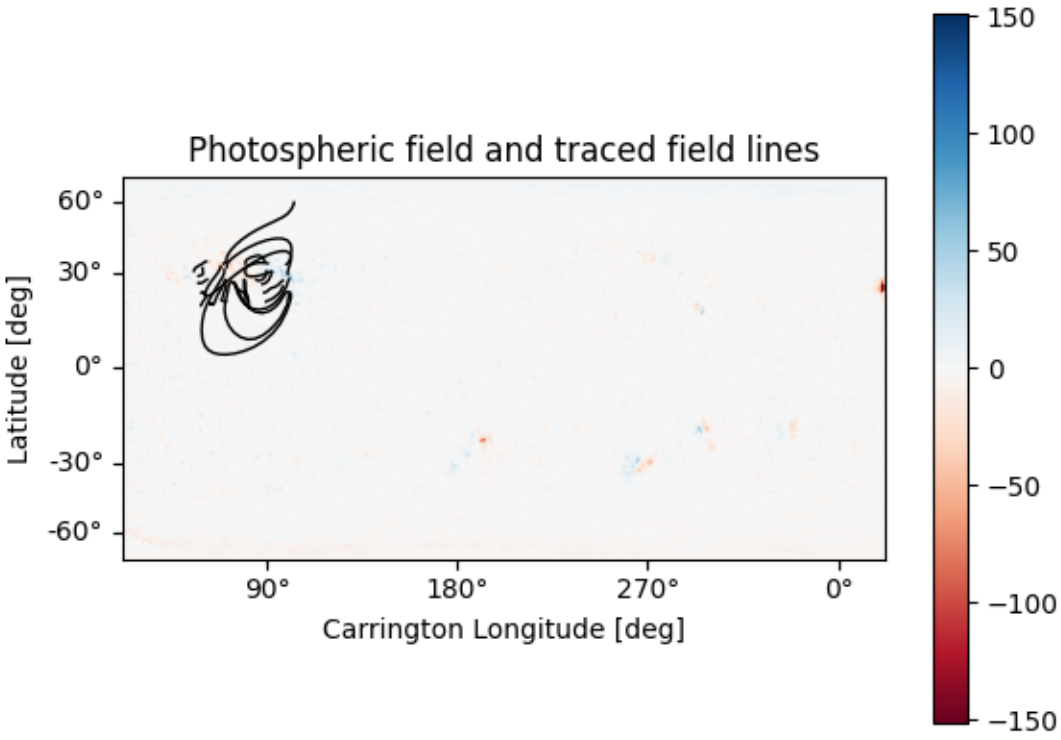
```
output = pfsspy.pfss(input)
```

Trace field lines from the footpoints defined above.

```
tracer = tracing.PythonTracer()
flines = tracer.trace(seeds, output)
```

Plot the input GONG magnetic field map, along with the traced mangetic field lines.

```
m = input.map
fig = plt.figure()
ax = plt.subplot(projection=m)
m.plot()
plt.colorbar()

for fline in flines:
    ax.plot_coord(fline.coords, color='black', linewidth=1)

# Set the axes limits. These limits have to be in pixel values
# ax.set_xlim(0, 180)
# ax.set_ylim(45, 135)
ax.set_title('Photospheric field and traced field lines')
```

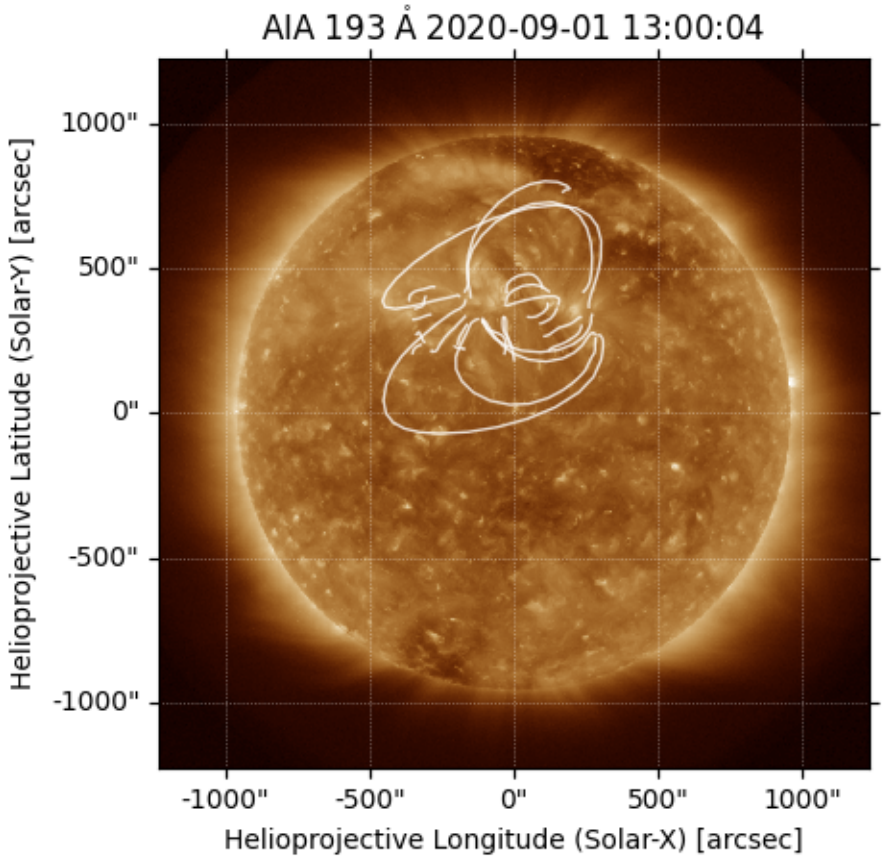


Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/pfsspy/envs/0.6.3/lib/python3.7/site-
↪packages/astropy/visualization/wcsaxes/core.py:211: MatplotlibDeprecationWarning:␣
↪Passing parameters norm and vmin/vmax simultaneously is deprecated since 3.3 and␣
↪will become an error two minor releases later. Please pass vmin/vmax directly to␣
↪the norm when creating it.
  return super().imshow(X, *args, origin=origin, **kwargs)

Text(0.5, 1.0, 'Photospheric field and traced field lines')
```

Plot the AIA map, along with the traced magnetic field lines. Inside the loop the field lines are converted to the AIA observer coordinate frame, and then plotted on top of the map.

```
fig = plt.figure()
ax = plt.subplot(1, 1, 1, projection=aia)
aia.plot(ax)
for fline in flines:
    ax.plot_coord(fline.coords, alpha=0.8, linewidth=1, color='white')

# ax.set_xlim(500, 900)
# ax.set_ylim(400, 800)
plt.show()

# sphinx_gallery_thumbnail_number = 5
```



Total running time of the script: (0 minutes 12.522 seconds)

2.1.2 Finding data

Examples showing how to find, download, and load magnetograms.

HMI data

How to search for HMI data.

This example shows how to search for, download, and load HMI data, using the `sunpy.net.Fido` interface. HMI data is available via. the Joint Stanford Operations Center (JSOC), and the radial magnetic field synoptic maps come in two sizes:

- 'hmi.Synoptic_Mr_720s': 3600 x 1440 in (lon, lat)
- 'hmi.mrsynop_small_720s': 720 x 360 in (lon, lat)

For more information on the maps, see the synoptic maps page on the JSOC site.

First import the required modules

```
from sunpy.net import Fido, attrs as a
import sunpy.map
```

Set up the search.

Note that for SunPy versions earlier than 2.0, a time attribute is needed to do the search, even if (in this case) it isn't used, as the synoptic maps are labelled by Carrington rotation number instead of time

```
time = a.Time('2010/01/01', '2010/01/01')
series = a.jsoc.Series('hmi.mrsynop_small_720s')
```

Do the search. This will return all the maps in the 'hmi_mrsynop_small_720s series.'

```
result = Fido.search(time, series)
print(result)
```

Out:

```
Results from 1 Provider:

139 Results from the JSOCClient:
     T_REC      TELESCOP  INSTRUME WAVELNTH CAR_ROT
--------------- -------- --------- -------- -------
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2097
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2098
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2099
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2100
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2101
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2102
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2103
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2104
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2105
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2106
            ...      ...       ...      ...     ...
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2225
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2226
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2227
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2228
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2229
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2230
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2231
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2232
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2233
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2234
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2235
Length = 139 rows
```

If we just want to download a specific map, we can specify a Carrington rotation number. In addition, downloading files from JSOC requires a notification email. If you use this code, please replace this email address with your own one, registered here: http://jsoc.stanford.edu/ajax/register_email.html

```
crot = a.jsoc.PrimeKey('CAR_ROT', 2210)
result = Fido.search(time, series, crot, a.jsoc.Notify("jsoc@cadair.com"))
print(result)
```

Out:

```
Results from 1 Provider:

1 Results from the JSOCClient:
     T_REC      TELESCOP  INSTRUME WAVELNTH CAR_ROT
```

(continues on next page)

(continued from previous page)

```
--------------- -------- --------- -------- -------
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2210
```

Download the files. This downloads files to the default sunpy download directory.

```
files = Fido.fetch(result)
print(files)
```

Out:

```
Export request pending. [id="JSOC_20201022_1119_X_IN", status=2]
Waiting for 0 seconds...
2 URLs found for download. Full request totalling 2MB

Files Downloaded:   0%|          | 0/2 [00:00<?, ?file/s]

hmi.mrsynop_small_720s.2210.synopMr.fits:   0%|          | 0.00/1.05M [00:00<?, ?B/
↪s][A


hmi.mrsynop_small_720s.2210.epts.fits:   0%|          | 0.00/1.05M [00:00<?, ?B/s][A[A

hmi.mrsynop_small_720s.2210.synopMr.fits:   0%|          | 100/1.05M [00:00<23:30,␣
↪741B/s][A


hmi.mrsynop_small_720s.2210.epts.fits:   0%|          | 100/1.05M [00:00<23:33, 740B/
↪s][A[A


hmi.mrsynop_small_720s.2210.epts.fits:   2%|1         | 20.9k/1.05M [00:00<16:11, 1.
↪05kB/s][A[A

hmi.mrsynop_small_720s.2210.synopMr.fits:   2%|1         | 19.4k/1.05M [00:00<16:11,␣
↪1.06kB/s][A


hmi.mrsynop_small_720s.2210.epts.fits:   6%|5         | 58.0k/1.05M [00:00<10:56, 1.
↪50kB/s][A[A

hmi.mrsynop_small_720s.2210.synopMr.fits:   5%|5         | 52.3k/1.05M [00:00<10:59,␣
↪1.51kB/s][A


hmi.mrsynop_small_720s.2210.epts.fits:  10%|#         | 108k/1.05M [00:00<07:16, 2.
↪15kB/s] [A[A

hmi.mrsynop_small_720s.2210.synopMr.fits:   9%|9         | 98.0k/1.05M [00:00<07:21,␣
↪2.15kB/s][A


hmi.mrsynop_small_720s.2210.epts.fits:  17%|#7        | 181k/1.05M [00:00<04:42, 3.
↪06kB/s][A[A

hmi.mrsynop_small_720s.2210.synopMr.fits:  17%|#6        | 175k/1.05M [00:00<04:44, 3.
↪06kB/s] [A
```

(continues on next page)

(continued from previous page)

```
hmi.mrsynop_small_720s.2210.epts.fits:  26%|##6       | 275k/1.05M [00:00<02:56, 4.
↪36kB/s][A[A

hmi.mrsynop_small_720s.2210.synopMr.fits:  27%|##6       | 278k/1.05M [00:00<02:55, 4.
↪37kB/s][A


hmi.mrsynop_small_720s.2210.epts.fits:  38%|###8      | 399k/1.05M [00:00<01:43, 6.
↪22kB/s][A[A

hmi.mrsynop_small_720s.2210.synopMr.fits:  40%|###9      | 415k/1.05M [00:00<01:41, 6.
↪23kB/s][A


hmi.mrsynop_small_720s.2210.epts.fits:  53%|#####2    | 552k/1.05M [00:01<00:55, 8.
↪87kB/s][A[A

hmi.mrsynop_small_720s.2210.synopMr.fits:  57%|#####6    | 594k/1.05M [00:01<00:50, 8.
↪89kB/s][A


hmi.mrsynop_small_720s.2210.epts.fits:  72%|#######1  | 749k/1.05M [00:01<00:23, 12.
↪6kB/s][A[A

hmi.mrsynop_small_720s.2210.synopMr.fits:  75%|#######5  | 789k/1.05M [00:01<00:20,␣
↪12.7kB/s][A


hmi.mrsynop_small_720s.2210.epts.fits:  92%|#########1| 961k/1.05M [00:01<00:04, 18.
↪0kB/s][A[A

hmi.mrsynop_small_720s.2210.synopMr.fits:  91%|#########1| 954k/1.05M [00:01<00:05,␣
↪18.0kB/s][A
Files Downloaded:  50%|#####     | 1/2 [00:01<00:01,  1.51s/file]
Files Downloaded: 100%|##########| 2/2 [00:01<00:00,  1.32file/s]
['/home/docs/sunpy/data/hmi.mrsynop_small_720s.2210.epts.fits', '/home/docs/sunpy/
↪data/hmi.mrsynop_small_720s.2210.synopMr.fits']



↪       [A                                                                      ␣



↪    [A[A                                                                       ␣
```

Read in a file. This will read in the first file downloaded to a sunpy Map object.

```
hmi_map = sunpy.map.Map(files[0])
print(hmi_map)
```

Out:

```
[[ 0.  0.  0. ...  0.  0.  0.]
 [15. 15. 15. ...  0.  0.  0.]
 [20. 20. 20. ... 10. 10. 10.]
```

(continues on next page)

(continued from previous page)

```
 ...
 [20. 20. 20. ... 20. 20. 20.]
 [20. 20. 20. ... 20. 20. 20.]
 [10. 10. 10. ... 20. 20. 20.]]
```

Total running time of the script: (0 minutes 13.601 seconds)

Parsing ADAPT Ensemble .fits files

Parse an ADAPT FITS file into a `sunpy.map.MapSequence`.

Necessary imports

```
import sunpy.map
import sunpy.io
import matplotlib.pyplot as plt
from matplotlib import gridspec

from pfsspy.sample_data import get_adapt_map
```

Load an example ADAPT fits file, utility stored in adapt_helpers.py

```
adapt_fname = get_adapt_map()
```

ADAPT synoptic magnetograms contain 12 realizations of synoptic magnetograms output as a result of varying model assumptions. See [here](https://www.swpc.noaa.gov/sites/default/files/images/u33/SWW_2012_Talk_04_27_2012_Arge.pdf))

Because the fits data is 3D, it cannot be passed directly to `sunpy.map.Map`, because this will take the first slice only and the other realizations are lost. We want to end up with a `sunpy.map.MapSequence` containing all these realiations as individual maps. These maps can then be individually accessed and PFSS solutions generated from them.

We first read in the fits file using `sunpy.io` :

```
adapt_fits = sunpy.io.fits.read(adapt_fname)
```

`adapt_fits` is a list of `HDPair` objects. The first of these contains the 12 realizations data and a header with sufficient information to build the `MapSequence`. We unpack this `HDPair` into a list of `(data,header)` tuples where `data` are the different adapt realizations.

```
data_header_pairs = [(map_slice, adapt_fits[0].header)
                                 for map_slice in adapt_fits[0].data]
```

Next, pass this list of tuples as the argument to `sunpy.map.Map` to create the map sequence :

```
adaptMapSequence = sunpy.map.Map(data_header_pairs, sequence=True)
```

`adapt_map_sequence` is now a list of our individual adapt realizations. Note the `.peek()` and ``.plot()` methods of `MapSequence` returns instances of `sunpy.visualization.MapSequenceAnimator` and `matplotlib.animation.FuncAnimation1`. Here, we generate a static plot accessing the individual maps in turn :

```
fig = plt.figure(figsize=(7, 8))
gs = gridspec.GridSpec(4, 3, figure=fig)
```

(continues on next page)

(continued from previous page)

```
for ii, aMap in enumerate(adaptMapSequence):
    ax = fig.add_subplot(gs[ii], projection=aMap)
    aMap.plot(axes=ax, cmap='bwr', vmin=-2, vmax=2, title=f"Realization {1+ii:02d}")
plt.tight_layout(pad=5, h_pad=2)
plt.show()
```

Total running time of the script: (0 minutes 0.000 seconds)

2.1.3 pfsspy utilities

Re-projecting from CAR to CEA

The pfsspy solver takes a cylindrical-equal-area (CEA) projected magnetic field map as input, which is equally spaced in sin(latitude). Some synoptic field maps are equally spaced in latitude however, which is a plate carée (CAR) projection.

This example shows how to use the *pfsspy.utils.car_to_cea* function to reproject a CAR projection to a CEA projection that pfsspy can take as input.

```
from pfsspy import sample_data
from pfsspy import utils
import matplotlib.pyplot as plt
```

Load a sample ADAPT map, which has a CAR projection

```
adapt_maps = utils.load_adapt(sample_data.get_adapt_map())
adapt_map_car = adapt_maps[0]
```

Out:

```
Files Downloaded:   0%|          | 0/1 [00:00<?, ?file/s]

adapt40311_03k012_202001010000_i00005600n1.fts.gz:   0%|          | 0.00/3.11M [00:00
↪<?, ?B/s][A

adapt40311_03k012_202001010000_i00005600n1.fts.gz:   0%|          | 100/3.11M [00:00
↪<1:53:29, 456B/s][A

adapt40311_03k012_202001010000_i00005600n1.fts.gz:   9%|8         | 265k/3.11M [00:00
↪<1:12:39, 652B/s][A

adapt40311_03k012_202001010000_i00005600n1.fts.gz:  31%|###       | 959k/3.11M [00:00
↪<38:26, 931B/s]  [A

adapt40311_03k012_202001010000_i00005600n1.fts.gz:  55%|#####4    | 1.70M/3.11M [00:00
↪<17:38, 1.33kB/s][A

adapt40311_03k012_202001010000_i00005600n1.fts.gz:  80%|########  | 2.50M/3.11M [00:00
↪<05:20, 1.90kB/s][A
Files Downloaded: 100%|##########| 1/1 [00:00<00:00,  1.02file/s]
Files Downloaded: 100%|##########| 1/1 [00:00<00:00,  1.02file/s]
```

Re-project into a CEA projection

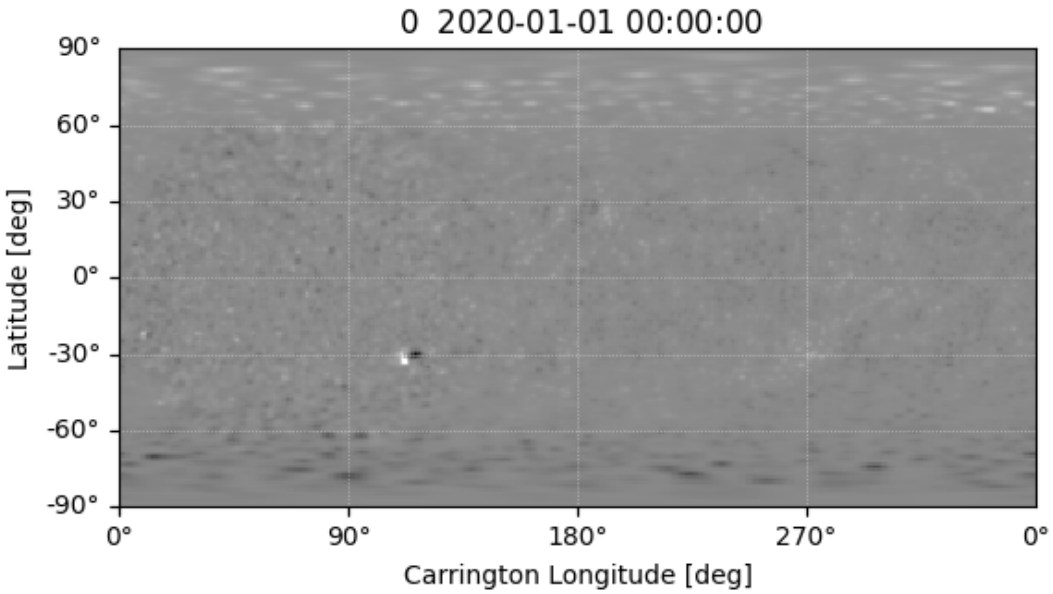
```
adapt_map_cea = utils.car_to_cea(adapt_map_car)
```

Out:

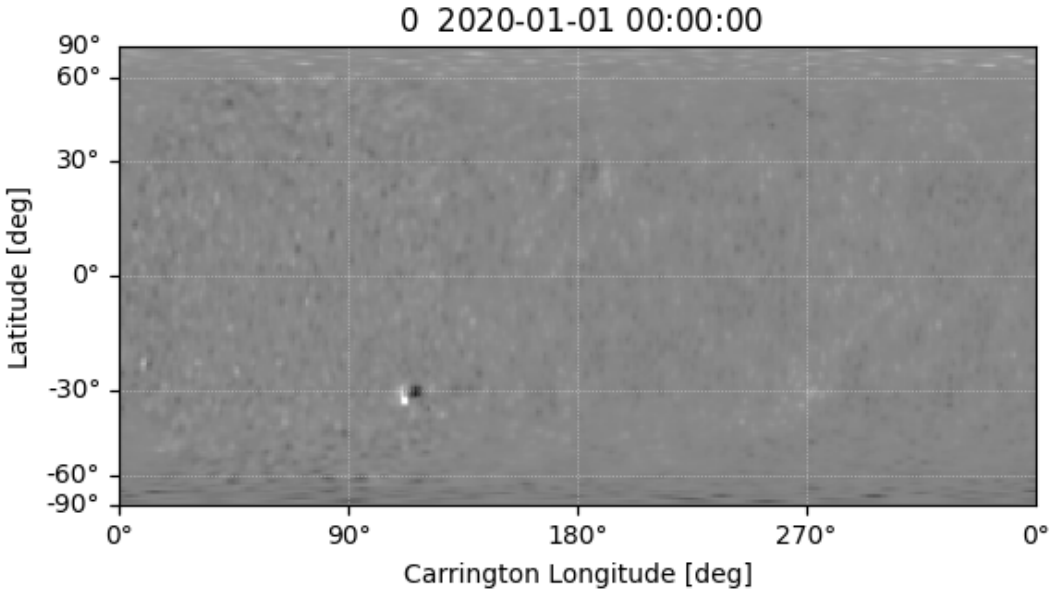
```
/home/docs/checkouts/readthedocs.org/user_builds/pfsspy/envs/0.6.3/lib/python3.7/site-
↪packages/sunpy/map/mapbase.py:838: SunpyUserWarning: Missing metadata for observer:␣
↪assuming Earth-based observer.

  warnings.warn(warning_message, SunpyUserWarning)
/home/docs/checkouts/readthedocs.org/user_builds/pfsspy/envs/0.6.3/lib/python3.7/site-
↪packages/sunpy/map/mapbase.py:838: SunpyUserWarning: Missing metadata for observer:␣
↪assuming Earth-based observer.

  warnings.warn(warning_message, SunpyUserWarning)
```

Plot the original map and the reprojected map

```
plt.figure()
adapt_map_car.plot()
plt.figure()
adapt_map_cea.plot()

plt.show()
```



-



-

Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/pfsspy/envs/0.6.3/lib/python3.7/site-
↪packages/sunpy/map/mapbase.py:838: SunpyUserWarning: Missing metadata for observer:␣
↪assuming Earth-based observer.

  warnings.warn(warning_message, SunpyUserWarning)
/home/docs/checkouts/readthedocs.org/user_builds/pfsspy/envs/0.6.3/lib/python3.7/site-
↪packages/sunpy/map/mapbase.py:838: SunpyUserWarning: Missing metadata for observer:␣
↪assuming Earth-based observer.

  warnings.warn(warning_message, SunpyUserWarning)
```

Total running time of the script: (0 minutes 2.876 seconds)

2.1.4 pfsspy information

Examples showing how the internals of pfsspy work.

pfsspy magnetic field grid

A plot of the grid corners, from which the magnetic field values are taken when tracing magnetic field lines.

Notice how the spacing becomes larger at the poles, and closer to the source surface. This is because the grid is equally spaced in $\cos\theta$ and $\log r$.

```
import numpy as np
import matplotlib.pyplot as plt
from pfsspy.grid import Grid
```

Define the grid spacings

```
ns = 15
nphi = 360
nr = 10
rss = 2.5
```

Create the grid

```
grid = Grid(ns, nphi, nr, rss)
```

Get the grid edges, and transform to r and theta coordinates

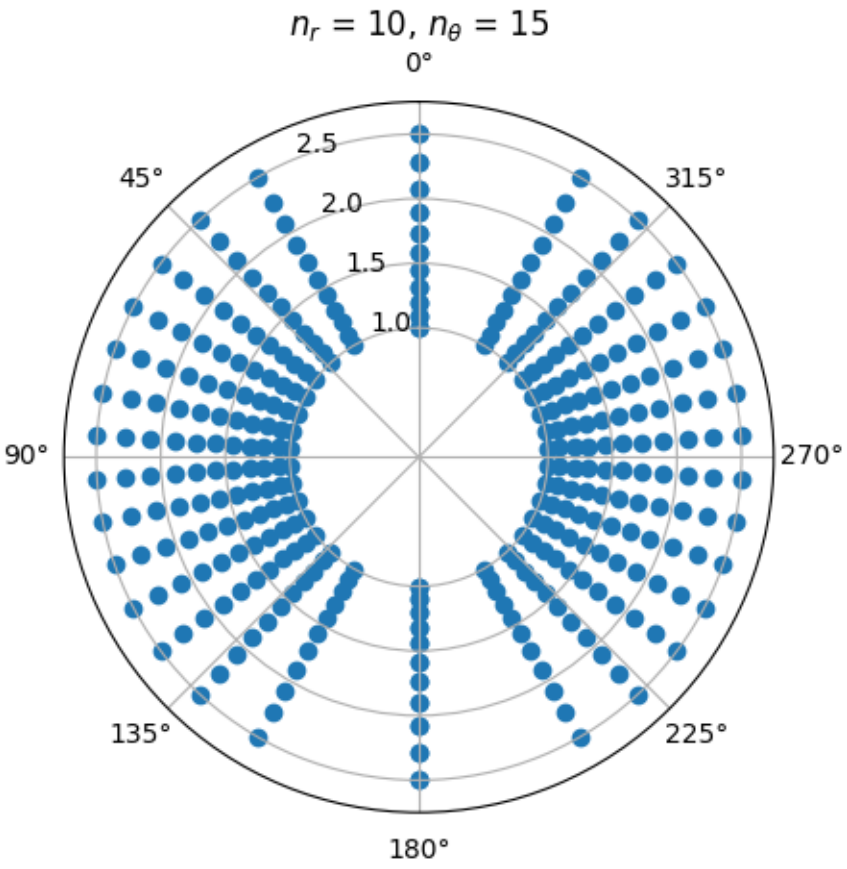
```
r_edges = np.exp(grid.rg)
theta_edges = np.arccos(grid.sg)
```

The corners of the grid are where lines of constant (r, theta) intersect, so meshgrid these together to get all the grid corners.

```
r_grid_points, theta_grid_points = np.meshgrid(r_edges, theta_edges)
```

Plot the resulting grid corners

```
fig = plt.figure()
ax = fig.add_subplot(projection='polar')

ax.scatter(theta_grid_points, r_grid_points)
ax.scatter(theta_grid_points + np.pi, r_grid_points, color='C0')

ax.set_ylim(0, 1.1 * rss)
ax.set_theta_zero_location('N')
ax.set_yticks([1, 1.5, 2, 2.5], minor=False)
ax.set_title('$n_{r}$ = ' f'{nr}, ' r'$n_{\theta}$ = ' f'{ns}')

plt.show()
```



Total running time of the script: (0 minutes 0.234 seconds)

Tracer performance

A quick script to compare the performance of the python and fortran tracers.

```
import timeit

import astropy.units as u
import astropy.coordinates
import numpy as np
import matplotlib.pyplot as plt
import sunpy.map

import pfsspy
```

Create a dipole map

```
ntheta = 180
nphi = 360
nr = 50
rss = 2.5

phi = np.linspace(0, 2 * np.pi, nphi)
theta = np.linspace(-np.pi / 2, np.pi / 2, ntheta)
```

(continues on next page)

(continued from previous page)

```
theta, phi = np.meshgrid(theta, phi)


def dipole_Br(r, theta):
    return 2 * np.sin(theta) / r**3


br = dipole_Br(1, theta)
br = sunpy.map.Map(br.T, pfsspy.utils.carr_cea_wcs_header('2010-01-01', br.shape))
pfss_input = pfsspy.Input(br, nr, rss)
pfss_output = pfsspy.pfss(pfss_input)
print('Computed PFSS solution')
```

Trace some field lines

```
seed0 = np.atleast_2d(np.array([1, 1, 0]))
tracers = [pfsspy.tracing.PythonTracer(),
           pfsspy.tracing.FortranTracer()]
nseeds = 2**np.arange(14)
times = [[], []]

for nseed in nseeds:
    print(nseed)
    seeds = np.repeat(seed0, nseed, axis=0)
    r, lat, lon = pfsspy.coords.cart2sph(seeds[:, 0], seeds[:, 1], seeds[:, 2])
    r = r * astropy.constants.R_sun
    lat = (lat - np.pi / 2) * u.rad
    lon = lon * u.rad
    seeds = astropy.coordinates.SkyCoord(lon, lat, r, frame=pfss_output.coordinate_
↪frame)

    for i, tracer in enumerate(tracers):
        if nseed > 64 and i == 0:
            continue

        t = timeit.timeit(lambda: tracer.trace(seeds, pfss_output), number=1)
        times[i].append(t)
```

Plot the results

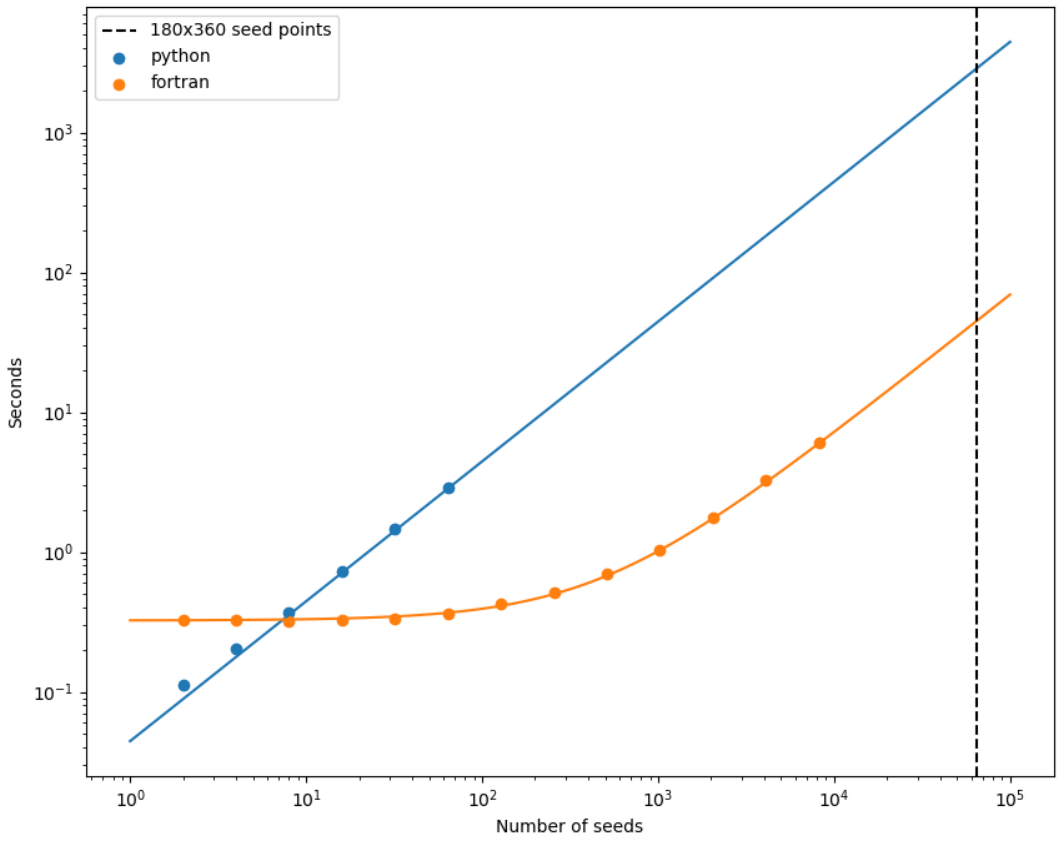
```
fig, ax = plt.subplots()
ax.scatter(nseeds[1:len(times[0])], times[0][1:], label='python')
ax.scatter(nseeds[1:], times[1][1:], label='fortran')

pydt = (times[0][4] - times[0][3]) / (nseeds[4] - nseeds[3])
ax.plot([1, 1e5], [pydt, 1e5 * pydt])

fort0 = times[1][1]
fordt = (times[1][-1] - times[1][-2]) / (nseeds[-1] - nseeds[-2])
ax.plot(np.logspace(0, 5, 100), fort0 + fordt * np.logspace(0, 5, 100))

ax.set_xscale('log')
ax.set_yscale('log')

ax.set_xlabel('Number of seeds')
ax.set_ylabel('Seconds')
```

(continues on next page)

(continued from previous page)

```
ax.axvline(180 * 360, color='k', linestyle='--', label='180x360 seed points')

ax.legend()
plt.show()
```

This shows the results of the above script, run on a 2014 MacBook pro with a 2.6 GHz Dual-Core Intel Core i5:



Total running time of the script: (0 minutes 0.000 seconds)

2.2 API reference

2.2.1 pfsspy Package

Functions

load_output(file)	Load a saved output file.
pfss(input)	Compute PFSS model.

load_output

pfsspy.**load_output**(*file*)
Load a saved output file.

Loads a file saved using *Output.save()*.

Parameters **file** (str, file, Path) – File to load.

Returns

Return type *Output*

pfss

pfsspy.**pfss**(*input*)
Compute PFSS model.

Extrapolates a 3D PFSS using an eigenfunction method in r, s, p coordinates, on the dumfric grid (equally spaced in $\rho = \ln(r/r_{sun})$, $s = \cos(\theta)$, and $p = \phi$).

Parameters **input** (*Input*) – Input parameters.

Returns **out**

Return type *Output*

Notes

In order to avoid numerical issues, the monopole term (which should be zero for a physical magnetic field anyway) is explicitly excluded from the solution.

The output should have zero current to machine precision, when computed with the DuMFriC staggered discretization.

Classes

Input(br, nr, rss)	Input to PFSS modelling.
Output(alr, als, alp, grid[, input_map])	Output of PFSS modelling.

Input

class pfsspy.**Input**(*br*, *nr*, *rss*)
Bases: object

Input to PFSS modelling.

> **Warning:** The input must be on a regularly spaced grid in ϕ and $s = \cos(\theta)$. See *pfsspy.grid* for more information on the coordinate system.

Parameters

- **br** (*sunpy.map.GenericMap*) – Boundary condition of radial magnetic field at the inner surface. Note that the data *must* have a cylindrical equal area projection.
- **nr** (*int*) – Number of cells in the radial direction to calculate the PFSS solution on.
- **rss** (*float*) – Radius of the source surface, as a fraction of the solar radius.

Attributes Summary

map	sunpy.map.GenericMap representation of the input.

Attributes Documentation

map
sunpy.map.GenericMap representation of the input.

Output

class pfsspy.**Output**(*alr*, *als*, *alp*, *grid*, *input_map=None*)
Bases: object

Output of PFSS modelling.

Parameters

- **alr** – Vector potential * grid spacing in radial direction.
- **als** – Vector potential * grid spacing in elevation direction.
- **alp** – Vector potential * grid spacing in azimuth direction.
- **grid** (Grid) – Grid that the output was caclulated on.
- **input_map** (*sunpy.map.GenericMap*) – The input map.

Notes

Instances of this class are intended to be created by *pfsspy.pfss*, and not by users.

Attributes Summary

bc	B on the centres of the cell faces.
bg	B as a (weighted) averaged on grid points.
bunit	Unit of the input map data.
coordinate_frame	The coordinate frame that the PFSS solution is in.
dtime	
source_surface_br	Br on the source surface.
source_surface_pils	Coordinates of the polarity inversion lines on the source surface.

Methods Summary

get_bvec(coords[, out_type])	Evaluate magnetic vectors in pfss model.
save(file)	Save the output to file.
trace(tracer, seeds)	**param tracer** Field line tracer.

Attributes Documentation

bc
B on the centres of the cell faces.

Returns

- *br*
- *btheta*
- *bphi*

bg
B as a (weighted) averaged on grid points.

Returns A `(nphi + 1, ns + 1, nrho + 1, 3)` shaped array. The last index gives the corodinate axis, 0 for Bphi, 1 for Bs, 2 for Brho.

Return type numpy.ndarray

bunit
`Unit` of the input map data.

coordinate_frame
The coordinate frame that the PFSS solution is in.

Notes

This is either a `HeliographicCarrington` or `HeliographicStonyhurst` frame, depending on the input map.

dtime

source_surface_br
Br on the source surface.

Returns

Return type `sunpy.map.GenericMap`

source_surface_pils
Coordinates of the polarity inversion lines on the source surface.

Notes

This is always returned as a list of coordinates, as in general there may be more than one polarity inversion line.

Methods Documentation

get_bvec(*coords*, *out_type='spherical'*)

Evaluate magnetic vectors in pfss model.

Method which takes an arbitrary astropy SkyCoord and returns a numpy array containing magnetic field vectors evaluated from the parent pfsspy.Output pfss model at the locations specified by the SkyCoords

Parameters

- **coords** (`astropy.SkyCoord`) – An arbitary point or set of points (length N >= 1) in the PFSS model domain (1Rs < r < Rss)
- **out_type** (*str, optional*) – Takes values 'spherical' (default) or 'cartesian' and specifies whether the output vector is in spherical coordinates (B_r,B_theta,B_phi) or cartesian (B_x,B_y,B_z)

Returns **bvec** – Magnetic field vectors at the requested locations ndarray.shape = (N,3), units nT)

Return type ndarray

Notes

The output coordinate system is defined by the input magnetogram with x-z plane equivalent to the plane containing the Carrington meridian (0 deg longitude)

The spherical coordinates follow the physics convention: https://upload.wikimedia.org/wikipedia/commons/thumb/4/4f/3D_Spherical.svg/240px-3D_Spherical.svg.png) Therefore the polar angle (theta) is the co-latitude, rather than the latitude, with range 0 (north pole) to 180 degrees (south pole)

The conversion which relates the spherical and cartesian coordinates is as follows:

$$B_R = sin\theta cos\phi B_x + sin\theta sin\phi B_y + cos\theta B_z$$

$$B_\theta = cos\theta cos\phi B_x + cos\theta sin\phi B_y - sin\theta B_z$$

$$B_\phi = -sin\phi B_x + cos\phi B_y$$

The above equations may be written as a (3x3) matrix and inverted to retrieve the inverse transformation (cartesian from spherical)

save(*file*)

Save the output to file.

This saves the required information to reconstruct an Output object in a compressed binary numpy file (see `numpy.savez_compressed()` for more information). The file extension is `.npz`, and is automatically added if not present.

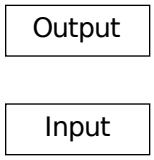
Parameters **file** (str, file, `Path`) – File to save to. If `.npz` extension isn't present it is added when saving the file.

trace(*tracer*, *seeds*)

Parameters

- **tracer** (tracing.Tracer) – Field line tracer.
- **seeds** (*astropy.coordinates.SkyCoord*) – Starting coordinates.

Class Inheritance Diagram



2.2.2 pfsspy.grid Module

Classes

Grid(ns, nphi, nr, rss)	Grid on which the pfsspy solution is calculated.

Grid

class pfsspy.grid.**Grid**(*ns*, *nphi*, *nr*, *rss*)
Bases: object

Grid on which the pfsspy solution is calculated.

Notes

The PFSS solution is calculated on a "strumfric" grid defined by

- $\rho = \log(r)$
- $s = \cos(\theta)$
- ϕ

where r, θ, ϕ are spherical cooridnates that have ranges

- $1 < r < r_{ss}$
- $0 < \theta < \pi$
- $0 < \phi < 2\pi$

Attributes Summary

dp	Cell size in phi.
dr	Cell size in log(r).
ds	Cell size in cos(theta).
pc	Location of the centre of cells in phi.
pg	Location of the edges of grid cells in phi.
rc	Location of the centre of cells in log(r).
rg	Location of the edges of grid cells in log(r).
sc	Location of the centre of cells in cos(theta).
sg	Location of the edges of grid cells in cos(theta).

Attributes Documentation

dp
Cell size in phi.

dr
Cell size in log(r).

ds
Cell size in cos(theta).

pc
Location of the centre of cells in phi.

pg
Location of the edges of grid cells in phi.

rc
Location of the centre of cells in log(r).

rg
Location of the edges of grid cells in log(r).

sc
Location of the centre of cells in cos(theta).

sg
Location of the edges of grid cells in cos(theta).

Class Inheritance Diagram

2.2.3 pfsspy.fieldline Module

Classes

ClosedFieldLines(field_lines)	A set of closed field lines.
FieldLine(x, y, z, output)	A single magnetic field line.
FieldLines(field_lines)	A collection of *FieldLine*.
OpenFieldLines(field_lines)	A set of open field lines.

ClosedFieldLines

class pfsspy.fieldline.**ClosedFieldLines**(*field_lines*)

Bases: *pfsspy.fieldline.FieldLines*

A set of closed field lines.

FieldLine

class pfsspy.fieldline.**FieldLine**(*x*, *y*, *z*, *output*)

Bases: object

A single magnetic field line.

Parameters

- **x** – Field line coordinates in cartesian coordinates.
- **y** – Field line coordinates in cartesian coordinates.
- **z** – Field line coordinates in cartesian coordinates.
- **output** (Output) – The PFSS output through which this field line was traced.

Attributes Summary

b_along_fline	The magnetic field vectors along the field line.
coords	Field line SkyCoord.
expansion_factor	Magnetic field expansion factor.
is_open	Returns True if one of the field line is connected to the solar surface and one to the outer boundary, False otherwise.
polarity	Magnetic field line polarity.
solar_footpoint	Solar surface magnetic field footpoint.
source_surface_footpoint	Solar surface magnetic field footpoint.

Attributes Documentation

b_along_fline
The magnetic field vectors along the field line.

coords
Field line `SkyCoord`.

expansion_factor
Magnetic field expansion factor.

The expansion factor is defnied as $(r_\odot^2 B_\odot)/(r_{ss}^2 B_{ss}))$

Returns **exp_fact** – Field line expansion factor.

Return type float

is_open
Returns `True` if one of the field line is connected to the solar surface and one to the outer boundary, `False` otherwise.

polarity
Magnetic field line polarity.

Returns **pol** – 0 if the field line is closed, otherwise sign(Br) of the magnetic field on the solar surface.

Return type int

solar_footpoint
Solar surface magnetic field footpoint.

This is the ends of the magnetic field line that lies on the solar surface.

Returns **footpoint**

Return type `SkyCoord`

Notes

For a closed field line, both ends lie on the solar surface. This method returns the field line pointing out from the solar surface in this case.

source_surface_footpoint
Solar surface magnetic field footpoint.

This is the ends of the magnetic field line that lies on the solar surface.

Returns **footpoint**

Return type `SkyCoord`

Notes

For a closed field line, both ends lie on the solar surface. This method returns the field line pointing out from the solar surface in this case.

FieldLines

class pfsspy.fieldline.**FieldLines**(*field_lines*)

Bases: object

A collection of *FieldLine*.

Parameters **field_lines** (list of *FieldLine*.) –

Attributes Summary

closed_field_lines	An *ClosedFieldLines* object containing open field lines.
connectivities	Field line connectivities.
expansion_factors	Expansion factors.
open_field_lines	An *OpenFieldLines* object containing open field lines.
polarities	Magnetic field line polarities.

Attributes Documentation

closed_field_lines
An *ClosedFieldLines* object containing open field lines.

connectivities
Field line connectivities. `1` for open, `0` for closed.

expansion_factors
Expansion factors. Set to NaN for closed field lines.

open_field_lines
An *OpenFieldLines* object containing open field lines.

polarities
Magnetic field line polarities. `0` for closed, otherwise sign(Br) on the solar surface.

OpenFieldLines

class pfsspy.fieldline.**OpenFieldLines**(*field_lines*)

Bases: *pfsspy.fieldline.FieldLines*

A set of open field lines.

Attributes Summary

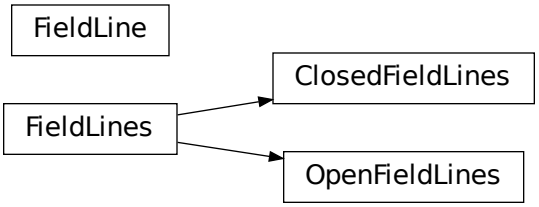
solar_feet	Coordinates of the solar footpoints.
source_surface_feet	Coordinates of the source suface footpoints.

Attributes Documentation

solar_feet
Coordinates of the solar footpoints.

source_surface_feet
Coordinates of the source suface footpoints.

Class Inheritance Diagram



2.2.4 pfsspy.tracing Module

Classes

FortranTracer([max_steps, step_size])	Tracer using Fortran code.
PythonTracer([atol, rtol])	Tracer using native python code.
Tracer()	Abstract base class for a streamline tracer.

FortranTracer

class pfsspy.tracing.**FortranTracer**(*max_steps=1000*, *step_size=0.01*)
Bases: *pfsspy.tracing.Tracer*

Tracer using Fortran code.

Parameters

- **max_steps** (*int*) – Maximum number of steps each streamline can take before stopping.
- **step_size** (*float*) – Step size as a fraction of cell size at the equator.

Notes

Because the stream tracing is done in spherical coordinates, there is a singularity at the poles (ie. $s = \pm 1$), which means seeds placed directly on the poles will not go anywhere.

Methods Summary

trace(seeds, output)	**param seeds** Coordinaes of the magnetic field seed points.
vector_grid(output)	Create a streamtracer.VectorGrid object from an *Output*.

Methods Documentation

trace (*seeds*, *output*)

Parameters

- **seeds** (*astropy.coordinates.SkyCoord*) – Coordinaes of the magnetic field seed points.
- **output** (pfsspy.Output) – pfss output.

Returns **streamlines** – Traced field lines.

Return type *FieldLines*

static vector_grid (*output*)

Create a streamtracer.VectorGrid object from an *Output*.

PythonTracer

class pfsspy.tracing.**PythonTracer** (*atol=0.0001*, *rtol=0.0001*)

Bases: *pfsspy.tracing.Tracer*

Tracer using native python code.

Uses scipy.integrate.solve_ivp, with an LSODA method.

Methods Summary

trace(seeds, output)	**param seeds** Coordinaes of the magnetic field seed points.

Methods Documentation

trace(*seeds*, *output*)

Parameters

- **seeds** (*astropy.coordinates.SkyCoord*) – Coordinaes of the magnetic field seed points.
- **output** (pfsspy.Output) – pfss output.

Returns **streamlines** – Traced field lines.

Return type *FieldLines*

Tracer

class pfsspy.tracing.**Tracer**

Bases: abc.ABC

Abstract base class for a streamline tracer.

Methods Summary

cartesian_to_coordinate()	Convert cartesian coordinate outputted by a tracer to a FieldLine object.
coords_to_xyz(seeds, output)	Given a set of astropy sky coordinates, transoform them to cartesian x, y, z coordinates.
trace(seeds, output)	**param seeds** Coordinaes of the magnetic field seed points.
validate_seeds(seeds)	Check that *seeds* has the right shape and is the correct type.

Methods Documentation

static cartesian_to_coordinate()

Convert cartesian coordinate outputted by a tracer to a FieldLine object.

static coords_to_xyz(*seeds*, *output*)

Given a set of astropy sky coordinates, transoform them to cartesian x, y, z coordinates.

Parameters

- **seeds** (*astropy.coordinates.SkyCoord*) –
- **output** (pfsspy.Output) –

abstract trace(*seeds*, *output*)

Parameters

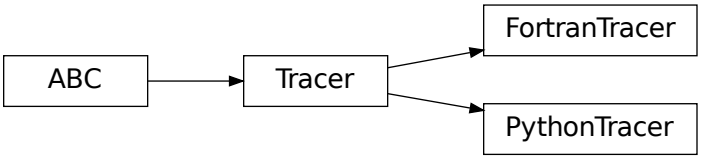
- **seeds** (*astropy.coordinates.SkyCoord*) – Coordinaes of the magnetic field seed points.
- **output** (pfsspy.Output) – pfss output.

Returns **streamlines** – Traced field lines.

Return type *FieldLines*

static validate_seeds(*seeds*)
Check that *seeds* has the right shape and is the correct type.

Class Inheritance Diagram



2.2.5 pfsspy.utils Module

Functions

car_to_cea(m[, method])	Reproject a plate-carée map in to a cylindrical-equal-area map.
carr_cea_wcs_header(dtime, shape)	Create a Carrington WCS header for a Cylindrical Equal Area (CEA) projection.
is_car_map(m[, error])	Returns `True` if *m* is in a plate carée projeciton.
is_cea_map(m[, error])	Returns `True` if *m* is in a cylindrical equal area projeciton.
is_full_sun_synoptic_map(m[, error])	Returns `True` if *m* is a synoptic map spanning the solar surface.
load_adapt(adapt_path)	Parse adapt .fts file as a `sunpy.map.MapSequence`

car_to_cea

pfsspy.utils.**car_to_cea**(*m*, *method='interp'*)
Reproject a plate-carée map in to a cylindrical-equal-area map.

The solver used in pfsspy requires a magnetic field map with values equally spaced in sin(lat) (ie. a CEA projection), but some maps are provided equally spaced in lat (ie. a CAR projection). This function reprojects a CAR map into a CEA map so it can be used with pfsspy.

Parameters

- **m** (*sunpy.map.GenericMap*) – Input map
- **method** (*str*) – Reprojection method to use. Can be `'interp'` (default), `'exact'`, or `'adaptive'`. See `reproject` for a description of the different methods. Note that different methods will give different results, and not all will conserve flux.

Returns **output_map** – Re-projected map. All metadata is preserved, apart from CTYPE{1,2} and CDELT2 which are updated to account for the new projection.

Return type sunpy.map.GenericMap

See also:

`reproject`

carr_cea_wcs_header

pfsspy.utils.**carr_cea_wcs_header**(*dtime*, *shape*)

Create a Carrington WCS header for a Cylindrical Equal Area (CEA) projection. See[1] for information on how this is constructed.

Parameters

- **dtime** (*datetime, None*) – Datetime to associate with the map.
- **shape** (*tuple*) – Map shape. The first entry should be number of points in longitude, the second in latitude.

References

is_car_map

pfsspy.utils.**is_car_map**(*m*, *error=False*)

Returns `True` if *m* is in a plate carée projeciton.

Parameters

- **m** (*sunpy.map.GenericMap*) –
- **error** (*bool*) – If `True`, raise an error if *m* is not a CAR projection.

is_cea_map

pfsspy.utils.**is_cea_map**(*m*, *error=False*)

Returns `True` if *m* is in a cylindrical equal area projeciton.

Parameters

- **m** (*sunpy.map.GenericMap*) –
- **error** (*bool*) – If `True`, raise an error if *m* is not a CEA projection.

[1] W. T. Thompson, "Coordinate systems for solar image data", https://doi.org/10.1051/0004-6361:20054262

is_full_sun_synoptic_map

pfsspy.utils.**is_full_sun_synoptic_map**(*m*, *error=False*)

Returns `True` if *m* is a synoptic map spanning the solar surface.

Parameters

- **m** (*sunpy.map.GenericMap*) –
- **error** (*bool*) – If `True`, raise an error if *m* does not span the whole solar surface.

load_adapt

pfsspy.utils.**load_adapt**(*adapt_path*)

Parse adapt .fts file as a `sunpy.map.MapSequence`

ADAPT magnetograms contain 12 realizations and their data attribute consists of a 3D data cube where each slice is the data corresponding to a separate realization of the magnetogram. This function loads the raw fits file and parses it to a `sunpy.map.MapSequence` object containing a `sunpy.map.Map` instance for each realization.

Parameters **adapt_path** (`str`) – Filepath corresponding to an ADAPT .fts file

Returns **adaptMapSequence**

Return type `sunpy.map.MapSequence`

2.3 Improving performance

2.3.1 numba

pfsspy automatically detects an installation of numba, which compiles some of the numerical code to speed up pfss calculations. To enable this simply install numba and use pfsspy as normal.

2.3.2 Streamline tracing

pfsspy has two streamline tracers: a pure python *pfsspy.tracing.PythonTracer* and a FORTRAN *pfsspy.tracing.FortranTracer*. The FORTRAN version is significantly faster, using the streamtracer package.

2.4 Changelog

2.4.1 0.6.3

This release contains the source for the accepted JOSS paper describing pfsspy.

2.4.2 0.6.2

This release includes several small fixes in response to a review of pfsspy for the Journal of Open Source Software. Thanks to Matthieu Ancellin and Simon Birrer for their helpful feedback!

- A permanent code of conduct file has been added to the repository.
- Information on how to contribute to pfsspy has been added to the docs.
- The example showing the performance of different magnetic field tracers has been fixed.
- The docs are now clearer about optional dependencies that can increase performance.
- The GONG example data has been updated due to updated data on the remote GONG server.

2.4.3 0.6.1

Bug fixes

- Fixed some messages in errors raised by functions in *pfsspy.utils*.

2.4.4 0.6.0

New features

- The *pfsspy.utils* module has been added, and contains various tools for loading and working with synoptic maps.
- *pfsspy.Output* has a new *bunit* property, which returns the `Unit` of the input map.
- Added *pfsspy.Output.get_bvec()*, to sample the magnetic field solution at arbitrary coordinates.
- Added the *pfsspy.fieldline.FieldLine.b_along_fline* property, to sample the magnetic field along a traced field line.
- Added a guide to the numerical methods used by pfsspy.

Breaking changes

- The `.al` property of *pfsspy.Output* is now private, as it is not intended for user access. If you *really* want to access it, use `._al` (but this is now private API and there is no guarantee it will stay or return the same thing in the future).
- A `ValueError` is now raised if any of the input data to *pfsspy.Input* is non-finite or NaN. Previously the PFSS computation would run fine, but the output would consist entirely of NaNs.

Behaviour changes

- The monopole term is now ignored in the PFSS calculation. Previously a non-zero (but small) monopole term would cause floating point precision issues, leading to a very noisy result. Now the monopole term is explicitly removed from the calculation. If your input has a non-zero mean value, pfsspy will issue a warning about this.
- The data downloaded by the examples is now automatically downloaded and cached with `sunpy.data.manager`. This means the files used for running the examples will be downloaded and stored in your `sunpy` data directory if they are required.
- The observer coordinate information in GONG maps is now automatically set to the location of Earth at the time in the map header.

Bug fixes

- The `date-obs` FITS keyword in GONG maps is now correctly populated.

2.4.5 0.5.3

- Improved descriptions in the AIA overplotting example.
- Fixed the 'date-obs' keyword in GONG metadata. Previously this just stored the date and not the time; now both the date and time are properly stored.
- Drastically sped up the calculation of source surface and solar surface magnetic field footpoints.

2.4.6 0.5.2

- Fixed a bug in the GONG synoptic map source where a map failed to load once it had already been loaded once.

2.4.7 0.5.1

- Fixed some calculations in `pfsspy.carr_cea_wcs_header`, and clarified in the docstring that the input shape must be in `[nlon, nlat]` order.
- Added validation to *`pfsspy.Input`* to check that the inputted map covers the whole solar surface.
- Removed ghost cells from *`pfsspy.Output.bc`*. This changes the shape of the returned arrays by one along some axes.
- Corrected the shape of *`pfsspy.Output.bg`* in the docstring.
- Added an example showing how to load ADAPT ensemble maps into a `CompositeMap`
- Sped up field line expansion factor calculations.

2.4.8 0.5.0

Changes to outputted maps

This release largely sees a transition to leveraging Sunpy Map objects. As such, the following changes have been made:

pfsspy.Input now *must* take a sunpy.map.GenericMap as an input boundary condition (as opposed to a numpy array). To convert a numpy array to a GenericMap, the helper function pfsspy.carr_cea_wcs_header can be used:

```
map_date = datetime(...)
br = np.array(...)
header = pfsspy.carr_cea_wcs_header(map_date, br.shape)

m = sunpy.map.Map((br, header))
pfss_input = pfsspy.Input(m, ...)
```

pfsspy.Output.source_surface_br now returns a GenericMap instead of an array. To get the data array use source_surface_br.data.

The new *pfsspy.Output.source_surface_pils* returns the coordinates of the polarity inversion lines on the source surface.

In favour of directly using the plotting functionality built into SunPy, the following plotting functionality has been removed:

- pfsspy.Input.plot_input. Instead *Input* has a new *map* property, which returns a SunPy map, which can easily be plotted using sunpy.map.GenericMap.plot.
- pfsspy.Output.plot_source_surface. A map of B_r on the source surface can now be obtained using *pfsspy.Output.source_surface_br*, which again returns a SunPy map.
- pfsspy.Output.plot_pil. The coordinates of the polarity inversion lines on the source surface can now be obtained using *pfsspy.Output.source_surface_pils*, which can then be plotted using ax.plot_coord(pil[0]) etc. See the examples section for an example.

Specifying tracing seeds

In order to make specifying seeds easier, they must now be a SkyCoord object. The coordinates are internally transformed to the Carrington frame of the PFSS solution, and then traced.

This should make specifying coordinates easier, as lon/lat/r coordinates can be created using:

```
seeds = astropy.coordinates.SkyCoord(lon, lat, r, frame=output.coordinate_frame)
```

To convert from the old x, y, z array used for seeds, do:

```
r, lat, lon = pfsspy.coords.cart2sph
r = r * astropy.constants.R_sun
lat = (lat - np.pi / 2) * u.rad
lon = lon * u.rad

seeds = astropy.coordinates.SkyCoord(lon, lat, r, frame=output.coordinate_frame)
```

Note that the latitude must be in the range $[-\pi/2, \pi/2]$.

GONG and ADAPT map sources

pfsspy now comes with built in `sunpy` map sources for GONG and ADAPT synoptic maps, which automatically fix some non-compliant FITS header values. To use these, just import `pfsspy` and load the .FITS files as normal with sunpy.

Tracing seeds

`pfsspy.tracing.Tracer` no longer has a `transform_seeds` helper method, which has been replaced by *`coords_to_xyz`* and `pfsspy.tracing.Tracer.xyz_to_coords`. These new methods convert between `SkyCoord` objects, and Cartesian xyz coordinates of the internal magnetic field grid.

2.4.9 0.4.3

- Improved the error thrown when trying to use :class`pfsspy.tracing.FotranTracer` without the `streamtracer` module installed.
- Fixed some layout issues in the documentation.

2.4.10 0.4.2

- Fix a bug where :class`pfsspy.tracing.FotranTracer` would overwrite the magnetic field values in an *`Output`* each time it was used.

2.4.11 0.4.1

- Reduced the default step size for the *`FortranTracer`* from 0.1 to 0.01 to give more resolved field lines by default.

2.4.12 0.4.0

New fortran field line tracer

`pfsspy.tracing` contains a new tracer, *`FortranTracer`*. This requires and uses the streamtracer package which does streamline tracing rapidly in python-wrapped fortran code. For large numbers of field lines this results in an ~50x speedup compared to the *`PythonTracer`*.

Changing existing code to use the new tracer is as easy as swapping out `tracer = pfsspy.tracer.PythonTracer()` for `tracer = pfsspy.tracer.FortranTracer()`. If you notice any issues with the new tracer, please report them at https://github.com/dstansby/pfsspy/issues.

Changes to field line objects

- `pfsspy.FieldLines` and `pfsspy.FieldLine` have moved to *pfsspy.fieldline.FieldLines* and *pfsspy.fieldline.FieldLine*.
- *FieldLines* no longer has `source_surface_feet` and `solar_feet` properties. Instead these have moved to the new *pfsspy.fieldline.OpenFieldLines* class. All the open field lines can be accessed from a *FieldLines* instance using the new *open_field_lines* property.

Changes to `Output`

- *pfsspy.Output.bg* is now returned as a 4D array instead of three 3D arrays. The final index now indexes the vector components; see the docstring for more information.

2.4.13 0.3.2

- Fixed a bug in `pfsspy.FieldLine.is_open`, where some open field lines were incorrectly calculated to be closed.

2.4.14 0.3.1

- Fixed a bug that incorrectly set closed line field polarities to -1 or 1 (instead of the correct value of zero).
- `FieldLine.footpoints` has been removed in favour of the new `pfsspy.FieldLine.solar_footpoint` and `pfsspy.FieldLine.source_surface_footpoint`. These each return a single footpoint. For a closed field line, see the API docs for further details on this.
- `pfsspy.FieldLines` has been added, as a convenience class to store a collection of field lines. This means convenience attributes such as `pfsspy.FieldLines.source_surface_feet` can be used, and their values are cached greatly speeding up repeated use.

2.4.15 0.3.0

- The API for doing magnetic field tracing has changed. The new *pfsspy.tracing* module contains *Tracer* classes that are used to perform the tracing. Code needs to be changed from:

```
fline = output.trace(x0)
```

  to:

```
tracer = pfsspy.tracing.PythonTracer()
tracer.trace(x0, output)
flines = tracer.xs
```

  Additionally `x0` can be a 2D array that contains multiple seed points to trace, taking advantage of the parallelism of some solvers.
- The `pfsspy.FieldLine` class no longer inherits from `SkyCoord`, but the `SkyCoord` coordinates are now stored in `pfsspy.FieldLine.coords` attribute.
- `pfsspy.FieldLine.expansion_factor` now returns `np.nan` instead of `None` if the field line is closed.

- `pfsspy.FieldLine` now has a `~pfsspy.FieldLine.footpoints` attribute that returns the footpoint(s) of the field line.

2.4.16 0.2.0

- *`pfsspy.Input`* and *`pfsspy.Output`* now take the optional keyword argument *dtime*, which stores the datetime on which the magnetic field measurements were made. This is then propagated to the *obstime* attribute of computed field lines, allowing them to be transformed in to coordinate systems other than Carrington frames.
- `pfsspy.FieldLine` no longer overrrides the SkyCoord `__init__`; this should not matter to users, as FieldLine objects are constructed internally by calling *`pfsspy.Output.trace`*

2.4.17 0.1.5

- `Output.plot_source_surface` now accepts keyword arguments that are given to Matplotlib to control the plotting of the source surface.

2.4.18 0.1.4

- Added more explanatory comments to the examples
- Corrected the dipole solution calculation
- Added `pfsspy.coords.sph2cart` to transform from spherical to cartesian coordinates.

2.4.19 0.1.3

- `pfsspy.Output.plot_pil` now accepts keyword arguments that are given to Matplotlib to control the style of the contour.
- `pfsspy.FieldLine.expansion_factor` is now cached, and is only calculated once if accessed multiple times.

2.5 Contributing to pfsspy

pfsspy is a community package, and contributions are welcome from anyone. This includes reporting bugs, requesting new features, along with writing code and improving documentation.

2.5.1 Reporting Issues

If you use pfsspy and stumble upon a problem, the best way to report it is by opening an issue on the GitHub issue tracker. This way we can help you work around the problem and hopefully fix the problem!

When reporting an issue, please try to provide a short description of the issue with a small code sample, this way we can attempt to reproduce the error. Also provide any error output generated when you encountered the issue, we can use this information to debug the issue.

2.5.2 Requesting new features

If there is functionality that is not currently available in pfsspy you can make a feature request on the issue page. Please add as much information as possible regarding the new feature you would like to see.

2.5.3 Improving documentaion

If something doesn't make sense in the online documentation, please open an issue. If you're comfortable fixing it, please open a pull request to the pfsspy repository.

2.5.4 Contributing code

All code contributions to pfsspy are welcome! If you would like to submit a contribution, please open a pull request at the pfsspy repository. It will then be reviewed, and if it is a useful addition merged into the main code branch.

If you are new to open source development, the instructions in these links might be useful for getting started:

- Astropy Dev Workflow
- Astropy Dev environment
- Astropy Pull Request Example

2.6 Numerical methods

For more information on the numerical methods used in the PFSS calculation see `this document`.

2.7 Synoptic map FITS conventions

FITS is the most common filetype used for the storing of solar images. On this page the FITS metadata conventions for synoptic maps are collected. All of this information can be found in, and is taken from, "Coordinate systems for solar image data (Thompson, 2005)".

Keyword	Output
CRPIX*n*	Reference pixel to subtract along axis *n*. Counts from 1 to N. Integer values refer to the centre of the pixel.
CRVAL*n*	Coordinate value of the reference pixel along axis *n*.
CDELT*n*	Pixel spacing along axis *n*.
CTYPE*n*	Coordinate axis label for axis *n*.
PV*i*_*m*	Additional parameters needed for some coordinate systems.

Note that *CROTAn* is ignored in this short guide.

2.7.1 Cylindrical equal area projection

In this projection, the latitude pixels are equally spaced in sin(latitude). The reference pixel has to be on the equator, to facilitate alignment with the solar rotation axis.

- CDELT2 is set to 180/pi times the pixel spacing in sin(latitude).
- CTYPE1 is either 'HGLN-CEA' or 'CRLN-CEA'.
- CTYPE2 is either 'HGLT-CEA' or 'CRLT-CEA'.
- PV*i*_1 is set to 1.
- LONPOLE is 0.

The abbreviations are "Heliographic Longitude - Cylindrical Equal Area" etc. If the system is heliographic the observer must also be defined in the metadata.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

R

S

T

V