pfsspy

pfsspy contributors

Aug 24, 2023

CONTENTS

1 Contents 3

2 Citing 79

Python Module Index 81

Index 83

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: From David Stansby, the lead author of *pfsspy*:

pfsspy has been archived, and is no longer developed - I no longer work in Solar Physics, and do not have time to maintain or support the package. *pfsspy* will probably continue working in the short term, but incompatibilities with dependencies will appear some point. The beauty of open source is that someone (maybe you!) can fork the code, and maintain, update, and improve it. If you do, I'd be greatful if you chose a new name for it and acknowledged the heritage of *pfsspy* in the new package.

Thanks to everyone who has contributed, whether through code or otherwise - this was a large part of my professional identity at the time, and I'm proud of the science it helped enable

---

CHAPTER

ONE

CONTENTS

1.1 Installing

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]
```

1.2 Examples

1.2.1 Using pfsspy

1.2.2 Finding data

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

1.2.3 Utilities

Useful code that doesn't involve doing a PFSS extrapolation.

1.2.4 Internals

1.2.5 Tests

Comparisons of the numerical output of pfsspy to analytic solutions.

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.

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

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

Load a GONG magnetic field map

```
gong_fname = get_gong_map()
gong_map = sunpy.map.Map(gong_fname)
```

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

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

pfsspy.mrzqs200901t1304c2234_022.fits.gz:   0%|          | 2.00/242k [00:00<4:34:41, 14.
↪7B/s]


Files Downloaded: 100%|| 1/1 [00:00<00:00,  2.66file/s]
Files Downloaded: 100%|| 1/1 [00:00<00:00,  2.65file/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 = 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

```
pfss_in = pfsspy.Input(gong_map, nrho, rss)
pfss_out = pfsspy.pfss(pfss_in)
```

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

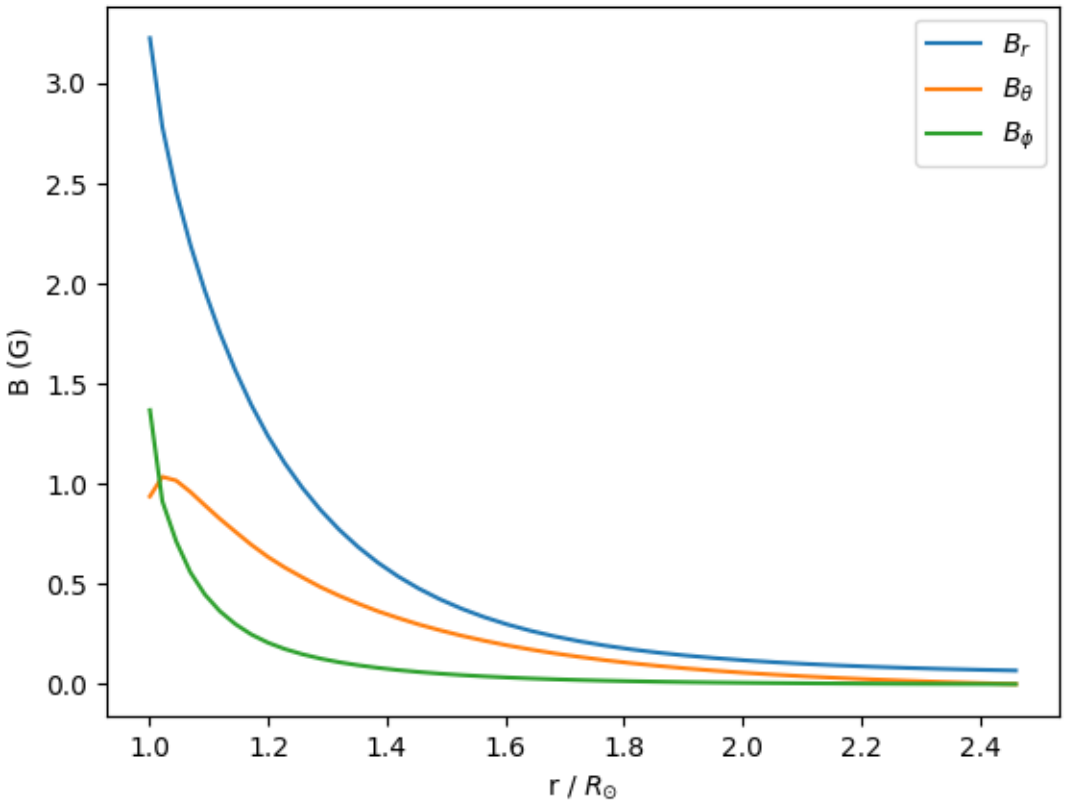
```
tracer = tracing.FortranTracer()
r = 1.2 * const.R_sun
lat = 70 * u.deg
lon = 0 * u.deg

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

```
INFO: Missing metadata for solar radius: assuming the standard radius of the photosphere.
↪ [sunpy.map.mapbase]
INFO: Missing metadata for solar radius: assuming the standard radius of the photosphere.
↪ [sunpy.map.mapbase]
```

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 7.126 seconds)

HMI PFSS solutions

Calculating a PFSS solution from a HMI synoptic map.

This example shows how to calcualte a PFSS solution from a HMI synoptic map. There are a couple of important things that this example shows:

- HMI maps have non-standard metadata, so this needs to be fixed
- HMI synoptic maps are very big (1440 x 3600), so need to be downsampled in order to calculate the PFSS solution in a reasonable time.

```
import os

import astropy.units as u
import matplotlib.pyplot as plt
import sunpy.map
from sunpy.net import Fido
from sunpy.net import attrs as a

import pfsspy
import pfsspy.utils
```

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.synoptic_mr_polfil_720s')
crot = a.jsoc.PrimeKey('CAR_ROT', 2210)
```

Do the search.

If you use this code, please replace this email address with your own one, registered here: http://jsoc.stanford.edu/ajax/register_email.html

```
result = Fido.search(time, series, crot,
                     a.jsoc.Notify(os.environ["JSOC_EMAIL"]))
files = Fido.fetch(result)
```

```
Export request pending. [id=JSOC_20230824_1951_X_IN, status=2]
Waiting for 0 seconds...
1 URLs found for download. Full request totalling 4MB

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

hmi.synoptic_mr_polfil_720s.2210.Mr_polfil.fits:   0%|          | 0.00/4.26M [00:00<?, ?
↪B/s]

hmi.synoptic_mr_polfil_720s.2210.Mr_polfil.fits:   0%|          | 1.02k/4.26M [00:00
↪<24:59, 2.84kB/s]
```

(continues on next page)

(continued from previous page)

```
hmi.synoptic_mr_polfil_720s.2210.Mr_polfil.fits:   2%|          | 99.0k/4.26M [00:00
↪<00:15, 267kB/s]

hmi.synoptic_mr_polfil_720s.2210.Mr_polfil.fits:  10%|          | 443k/4.26M [00:00<00:03,
↪ 1.09MB/s]

hmi.synoptic_mr_polfil_720s.2210.Mr_polfil.fits:  43%|     | 1.82M/4.26M [00:00<00:00, 4.
↪39MB/s]


Files Downloaded: 100%|| 1/1 [00:01<00:00,  1.17s/file]
Files Downloaded: 100%|| 1/1 [00:01<00:00,  1.17s/file]
```

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('Data shape: ', hmi_map.data.shape)
```

```
Data shape:  (1440, 3600)
```

Since this map is far to big to calculate a PFSS solution quickly, lets resample it down to a smaller size.

```
hmi_map = hmi_map.resample([360, 180] * u.pix)
print('New shape: ', hmi_map.data.shape)
```

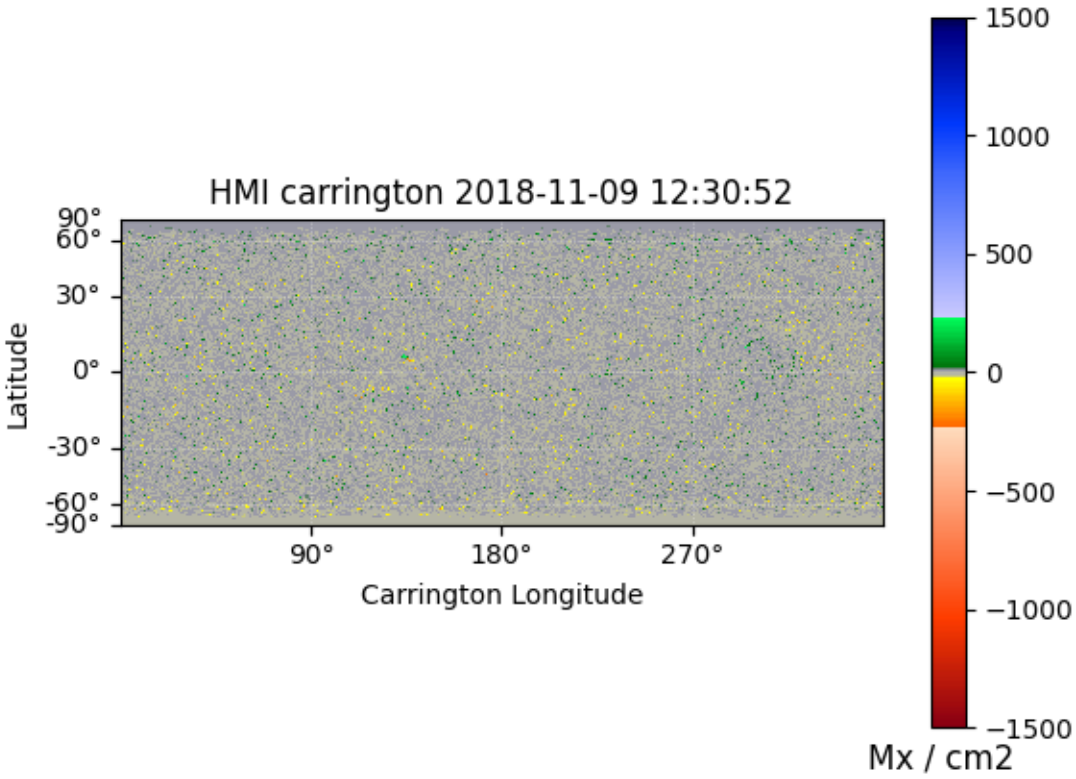
```
New shape:  (180, 360)
```

Now calculate the PFSS solution

```
nrho = 35
rss = 2.5
pfss_in = pfsspy.Input(hmi_map, nrho, rss)
pfss_out = pfsspy.pfss(pfss_in)
```

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

```
ss_br = pfss_out.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(pfss_out.source_surface_pils[0])
# Plot formatting
plt.colorbar()
ax.set_title('Source surface magnetic field')

plt.show()
```




```
/home/docs/checkouts/readthedocs.org/user_builds/pfsspy/envs/latest/lib/python3.10/site-
↪packages/pfsspy/output.py:95: UserWarning: Could not parse unit string "Mx/cm^2" as a␣
↪valid FITS unit.
See https://fits.gsfc.nasa.gov/fits_standard.html for the FITS unit standards.
  warnings.warn(f'Could not parse unit string "{unit_str}" as a valid FITS unit.\n'
INFO: Missing metadata for solar radius: assuming the standard radius of the photosphere.
↪ [sunpy.map.mapbase]
/home/docs/checkouts/readthedocs.org/user_builds/pfsspy/envs/latest/lib/python3.10/site-
↪packages/sunpy/map/mapbase.py:628: SunpyMetadataWarning: Missing metadata for␣
↪observer: assuming Earth-based observer.
For frame 'heliographic_stonyhurst' the following metadata is missing: dsun_obs,hglt_obs,
↪hgln_obs
For frame 'heliographic_carrington' the following metadata is missing: crlt_obs,dsun_obs,
↪crln_obs

  obs_coord = self.observer_coordinate
/home/docs/checkouts/readthedocs.org/user_builds/pfsspy/envs/latest/lib/python3.10/site-
↪packages/pfsspy/output.py:95: UserWarning: Could not parse unit string "Mx/cm^2" as a␣
↪valid FITS unit.
See https://fits.gsfc.nasa.gov/fits_standard.html for the FITS unit standards.
  warnings.warn(f'Could not parse unit string "{unit_str}" as a valid FITS unit.\n'
```

Total running time of the script: (0 minutes 19.317 seconds)

Open/closed field map

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

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

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

Load a GONG magnetic field map

```
gong_fname = get_gong_map()
gong_map = sunpy.map.Map(gong_fname)
```

Set the model parameters

```
nrho = 40
rss = 2.5
```

Construct the input, and calculate the output solution

```
pfss_in = pfsspy.Input(gong_map, nrho, rss)
pfss_out = pfsspy.pfss(pfss_in)
```

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 = 45
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=pfss_out.coordinate_frame)
```

```
INFO: Missing metadata for solar radius: assuming the standard radius of the photosphere.
↪ [sunpy.map.mapbase]
INFO: Missing metadata for solar radius: assuming the standard radius of the photosphere.
↪ [sunpy.map.mapbase]
```

Trace the field lines

```
print('Tracing field lines...')
tracer = tracing.FortranTracer(max_steps=2000)
```

(continues on next page)

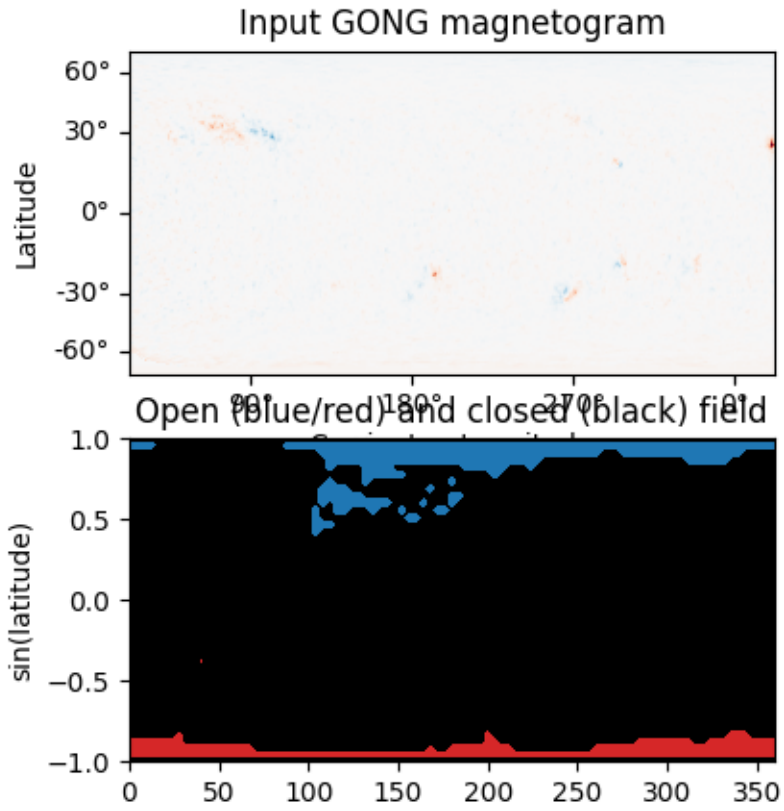
(continued from previous page)

```
field_lines = tracer.trace(seeds, pfss_out)
print('Finished tracing field lines')
```

```
Tracing field lines...
/home/docs/checkouts/readthedocs.org/user_builds/pfsspy/envs/latest/lib/python3.10/site-
↪packages/pfsspy/tracing.py:180: UserWarning: At least one field line ran out of steps␣
↪during tracing.
You should probably increase max_steps (currently set to 2000) and try again.
  warnings.warn(
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 = pfss_in.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)

plt.show()
```



Total running time of the script: (0 minutes 18.806 seconds)

Dipole source solution

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

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

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.

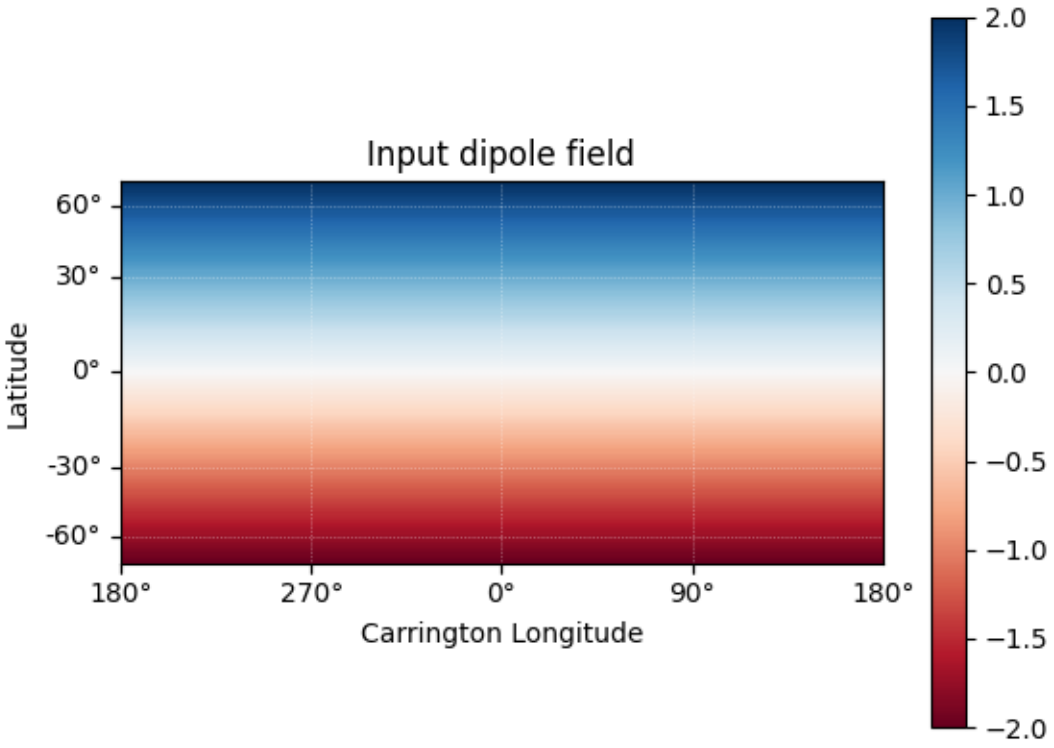
```
nrho = 30
rss = 2.5
```

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))
pfss_in = pfsspy.Input(input_map, nrho, rss)
```

Using the Input object, plot the input field

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



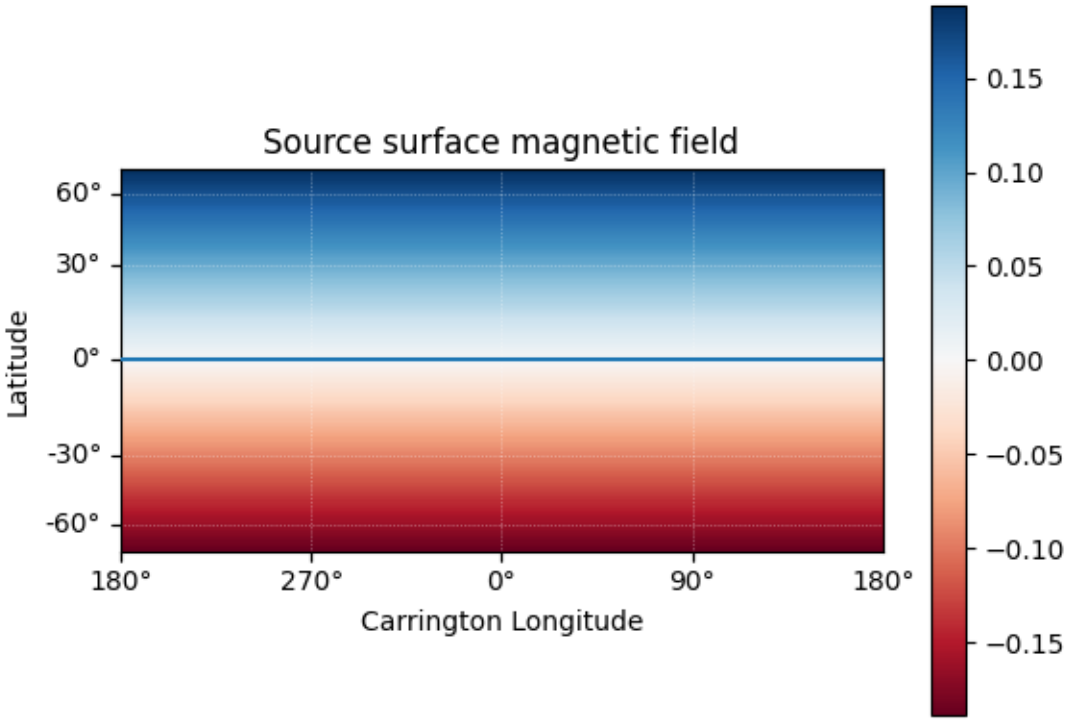

```
Text(0.5, 1.0, 'Input dipole field')
```

Now calculate the PFSS solution.

```
pfss_out = pfsspy.pfss(pfss_in)
```

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

```
ss_br = pfss_out.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(pfss_out.source_surface_pils[0])
# Plot formatting
plt.colorbar()
ax.set_title('Source surface magnetic field')
```




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

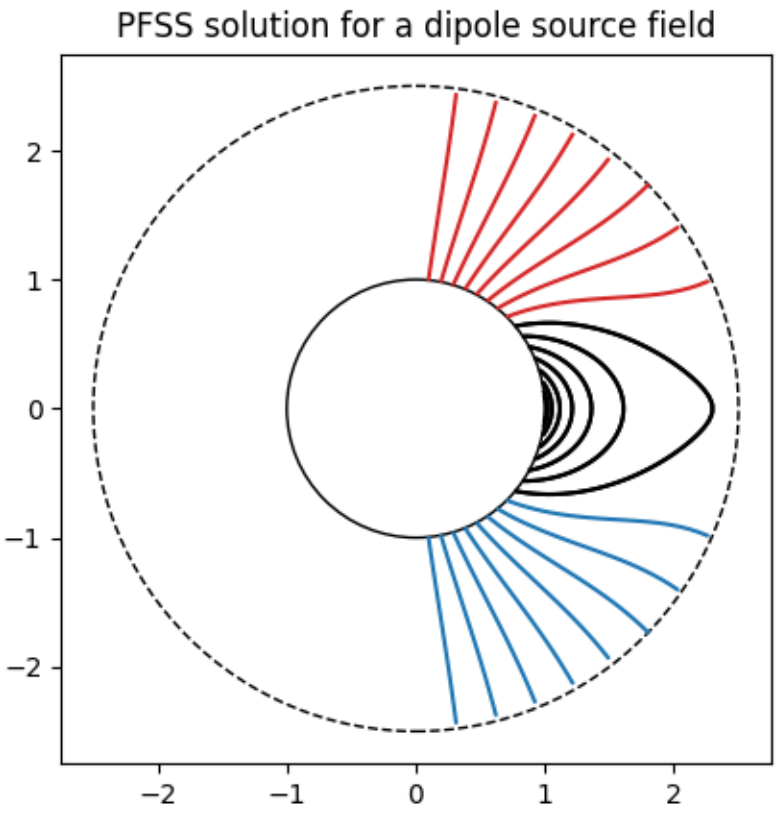
Finally, using the 3D magnetic field solution we can trace some field lines. In this case 32 points equally spaced in theta are chosen and traced from the source surface outwards.

```
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=pfss_out.coordinate_frame)

tracer = pfsspy.tracing.FortranTracer()
field_lines = tracer.trace(seeds, pfss_out)

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)
```

(continues on next page)

(continued from previous page)

```
# 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), pfss_in.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 7.206 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.

```
import os

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

(continues on next page)

(continued from previous page)

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

Load a GONG magnetic field map

```
gong_fname = get_gong_map()
gong_map = sunpy.map.Map(gong_fname)
```

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.

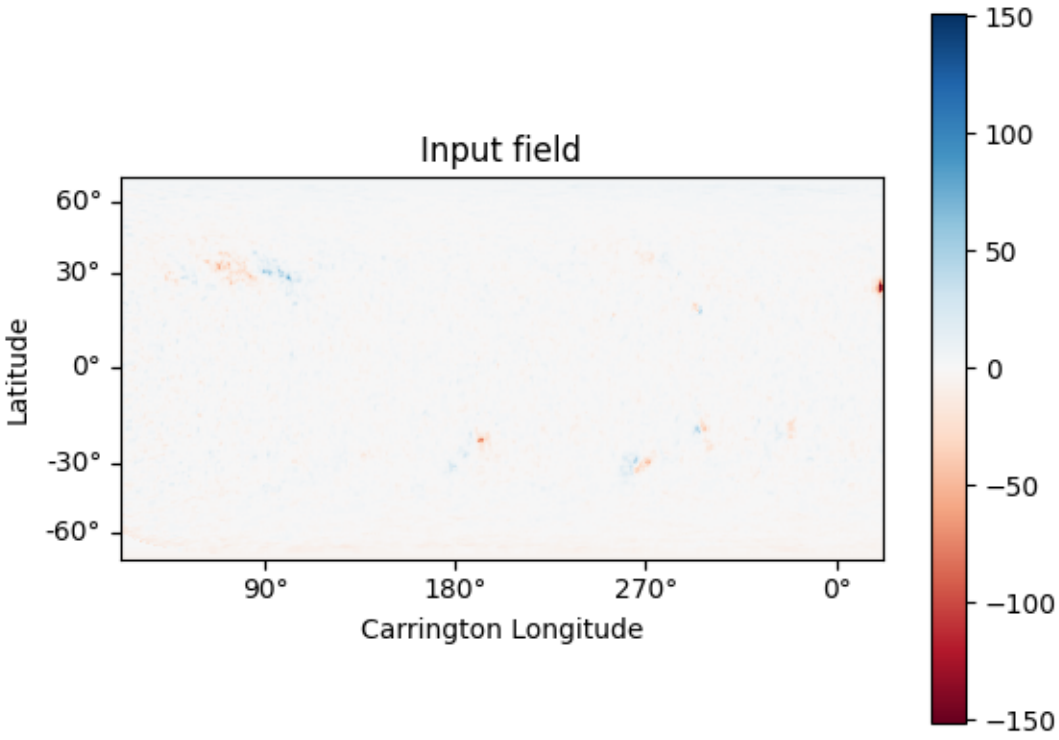
```
nrho = 25
rss = 2.5
```

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

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

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

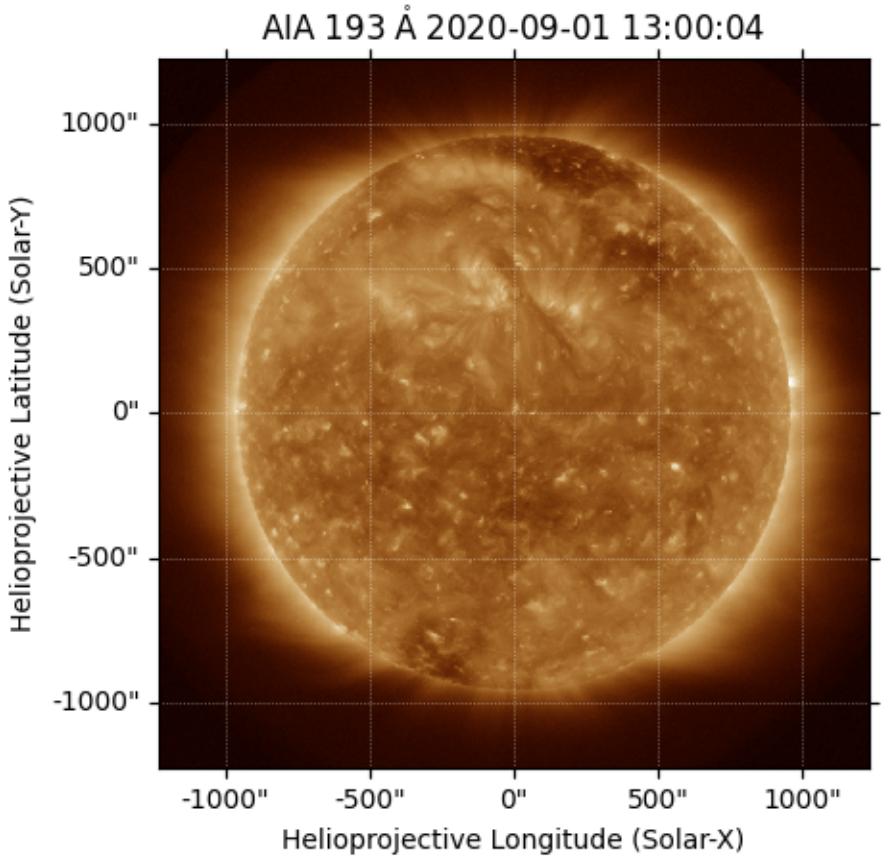
```
m = pfss_in.map
fig = plt.figure()
ax = plt.subplot(projection=m)
m.plot()
plt.colorbar()
ax.set_title('Input field')
```




```
INFO: Missing metadata for solar radius: assuming the standard radius of the photosphere.
↪ [sunpy.map.mapbase]
INFO: Missing metadata for solar radius: assuming the standard radius of the photosphere.
↪ [sunpy.map.mapbase]

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.

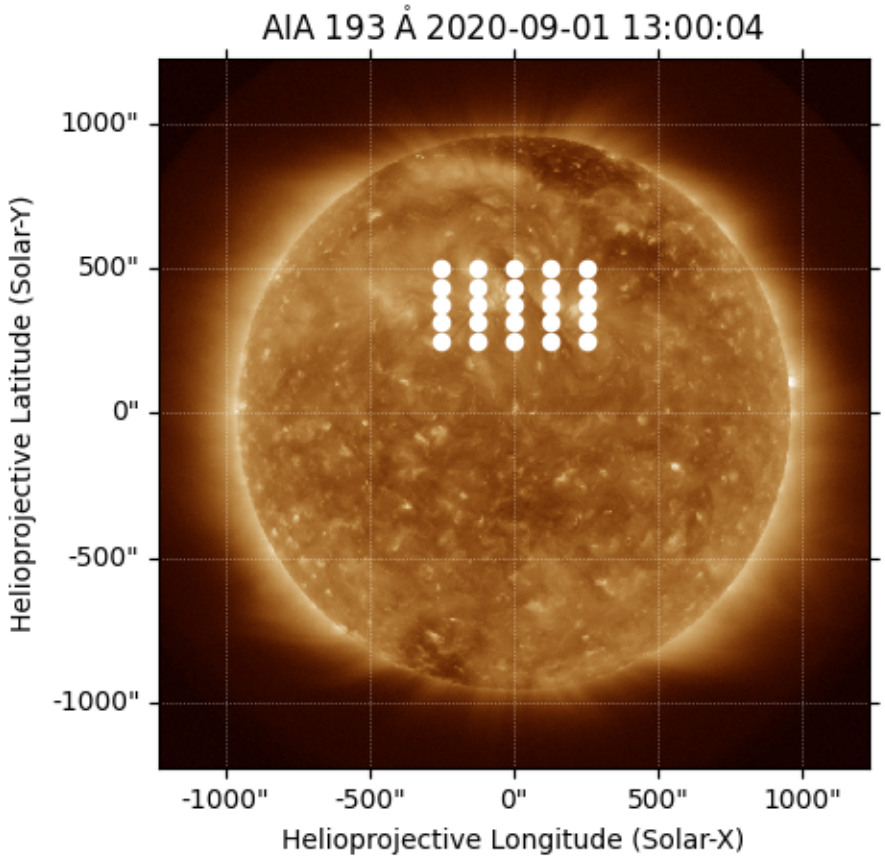
```
ax = plt.subplot(1, 1, 1, projection=aia)
aia.plot(ax)
```




```
<matplotlib.image.AxesImage object at 0x7f1cec711b70>
```

Now we construct a 5 x 5 grid of footpoitns to trace some magnetic field lines from. These coordinates are defined in the helioprojective frame of the AIA image

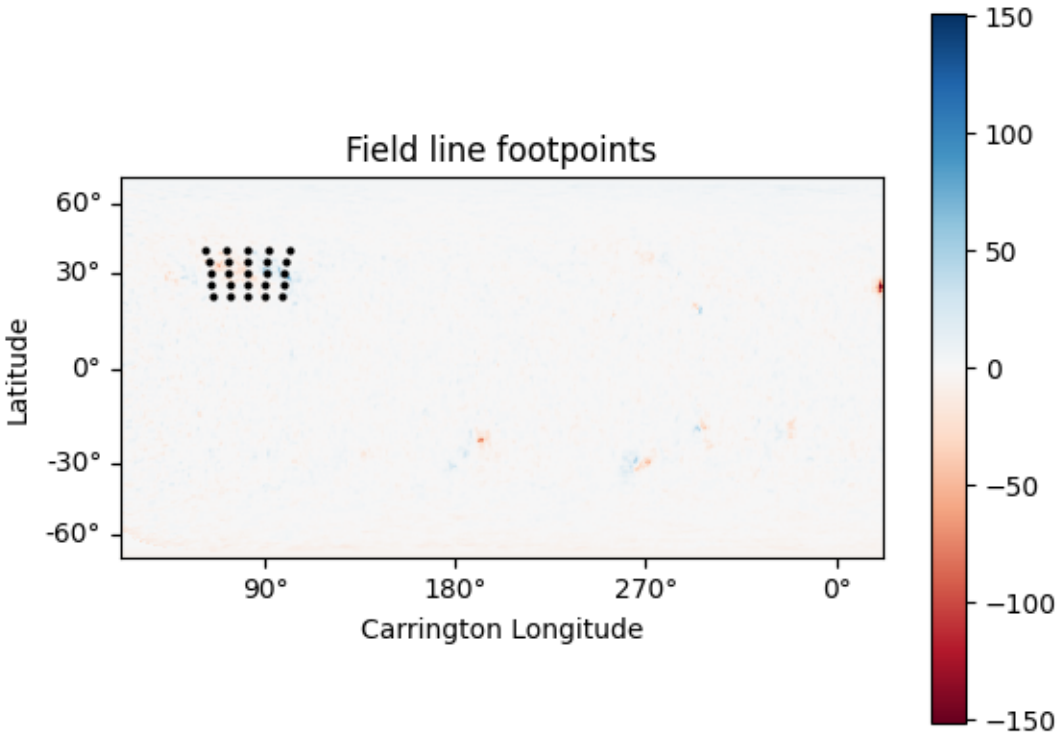
```
hp_lon = np.linspace(-250, 250, 5) * u.arcsec
hp_lat = np.linspace(250, 500, 5) * u.arcsec
# Make a 2D grid from these 1D points
lon, lat = np.meshgrid(hp_lon, hp_lat)
seeds = SkyCoord(lon.ravel(), lat.ravel(),
                 frame=aia.coordinate_frame)
fig = plt.figure()
ax = plt.subplot(projection=aia)
aia.plot(axes=ax)
ax.plot_coord(seeds, color='white', marker='o', linewidth=0)
```




```
[<matplotlib.lines.Line2D object at 0x7f1cf2c1df00>]
```

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

```
m = pfss_in.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)
```




```
(0.0, 179.5)
```

Compute the PFSS solution from the GONG magnetic field input

```
pfss_out = pfsspy.pfss(pfss_in)
```

Trace field lines from the footpoints defined above.

```
tracer = tracing.FortranTracer()
flines = tracer.trace(seeds, pfss_out)
```

```
/home/docs/checkouts/readthedocs.org/user_builds/pfsspy/envs/latest/lib/python3.10/site-
↪packages/pfsspy/tracing.py:180: UserWarning: At least one field line ran out of steps␣
↪during tracing.
You should probably increase max_steps (currently set to auto) and try again.
  warnings.warn(
```

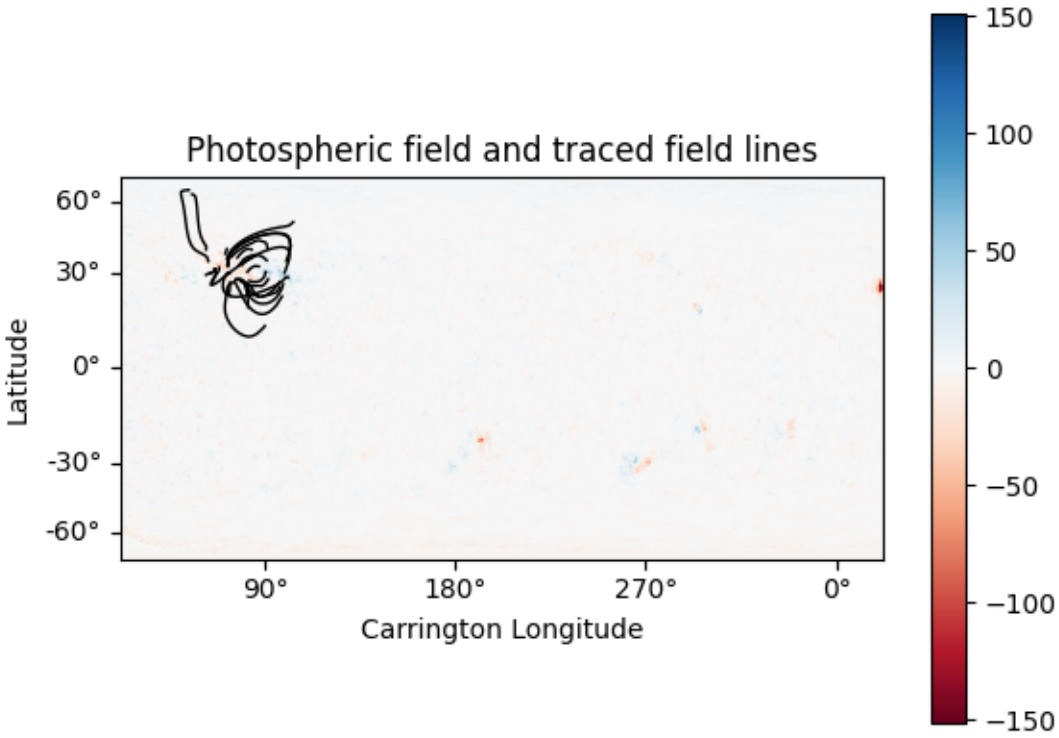
Plot the input GONG magnetic field map, along with the traced mangetic field lines.

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

(continues on next page)

(continued from previous page)

```
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')
```




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

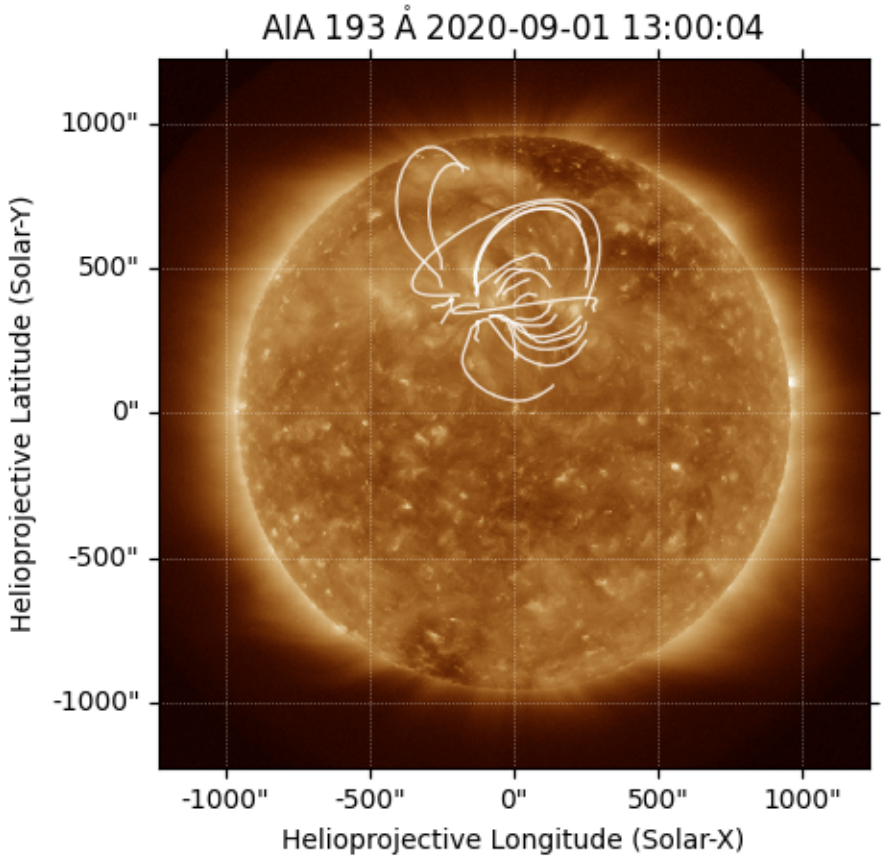
Plot the AIA map, along with the traced magnetic field lines. Inside the loop the field lines are converted to the AIA observer coordinate frame, and then plotted on top of the map.

```
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)
```

(continues on next page)

(continued from previous page)

```
plt.show()

# sphinx_gallery_thumbnail_number = 5
```



Total running time of the script: (0 minutes 12.252 seconds)

GONG PFSS extrapolation

Calculating PFSS solution for a GONG synoptic magnetic field map.

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

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

Load a GONG magnetic field map

```
gong_fname = get_gong_map()
gong_map = sunpy.map.Map(gong_fname)
```

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.

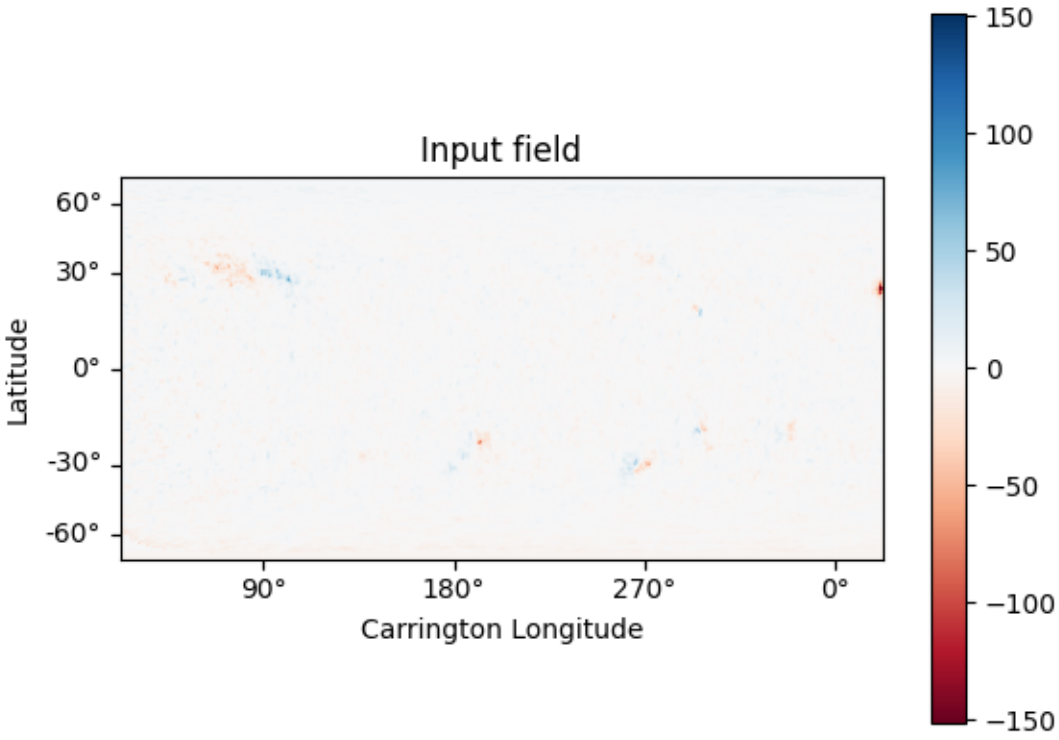
```
nrho = 35
rss = 2.5
```

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

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



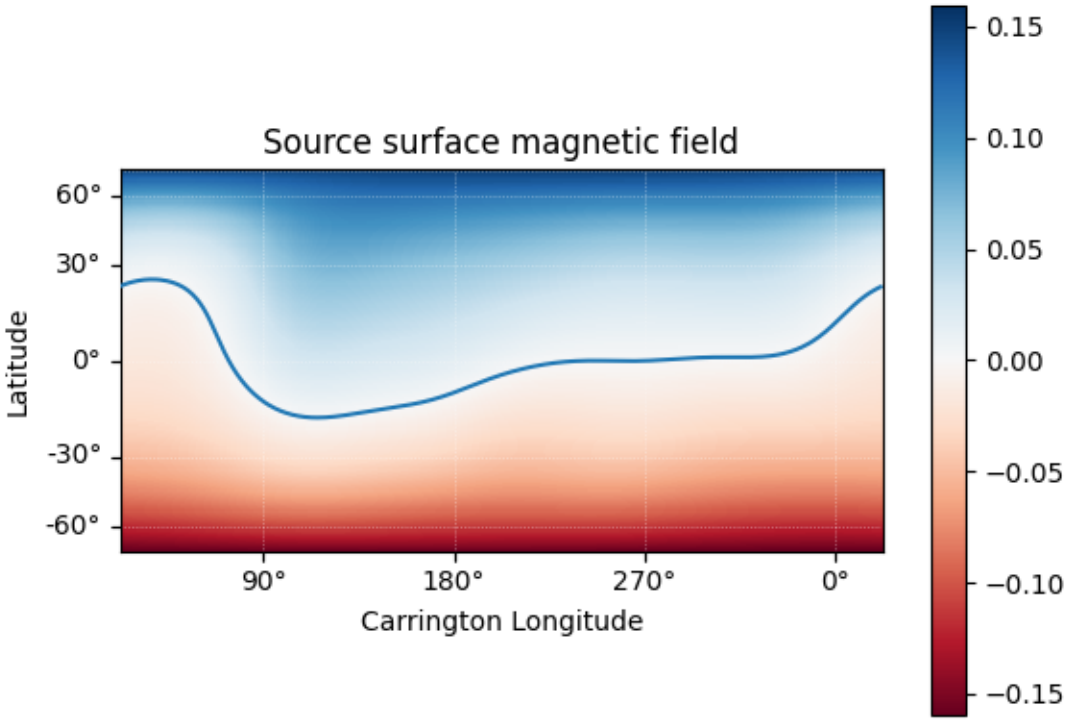

```
INFO: Missing metadata for solar radius: assuming the standard radius of the photosphere.
↪ [sunpy.map.mapbase]
INFO: Missing metadata for solar radius: assuming the standard radius of the photosphere.
↪ [sunpy.map.mapbase]
```

Now calculate the PFSS solution

```
pfss_out = pfsspy.pfss(pfss_in)
```

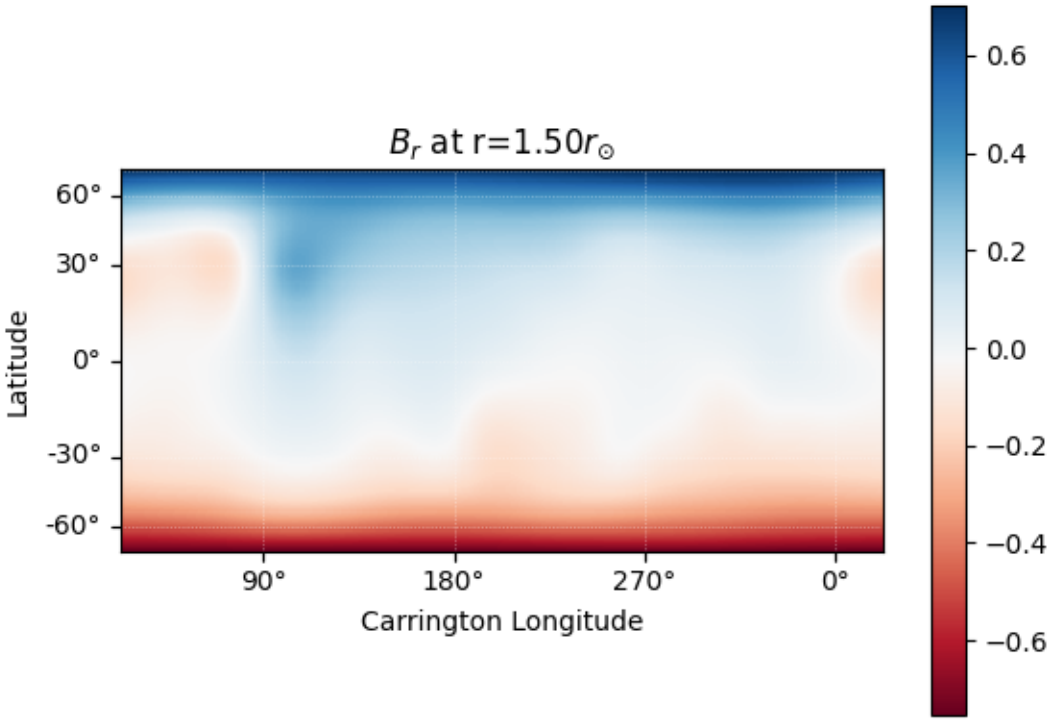
Using the Output object we can plot the source surface field, and the polarity inversion line.

```
ss_br = pfss_out.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(pfss_out.source_surface_pils[0])
# Plot formatting
plt.colorbar()
ax.set_title('Source surface magnetic field')
set_axes_lims(ax)
```



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

# Create the figure and axes
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)
```



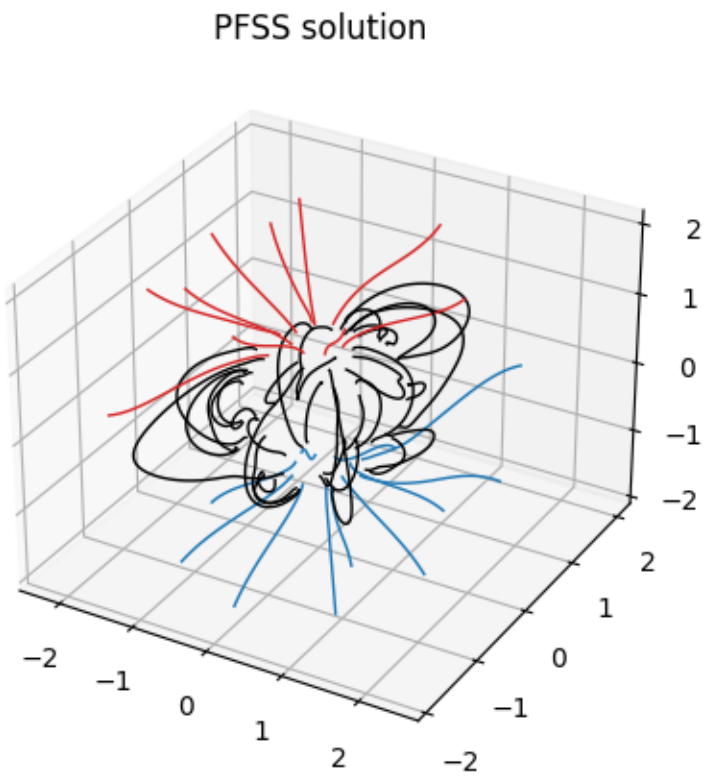
Finally, using the 3D magnetic field solution we can trace some field lines. In this case 64 points equally gridded in theta and phi are chosen and traced from the source surface outwards.

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

tracer = tracing.FortranTracer()
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=pfss_out.coordinate_frame)

field_lines = tracer.trace(seeds, pfss_out)

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,
```

(continues on next page)

(continued from previous page)

```
            color=color, linewidth=1)

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

# sphinx_gallery_thumbnail_number = 4
```




```
/home/docs/checkouts/readthedocs.org/user_builds/pfsspy/envs/latest/lib/python3.10/site-
↪packages/pfsspy/tracing.py:180: UserWarning: At least one field line ran out of steps␣
↪during tracing.
You should probably increase max_steps (currently set to auto) and try again.
  warnings.warn(
```

Total running time of the script: (0 minutes 7.976 seconds)

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).

The polar filled radial magnetic field synoptic maps are obtained using the 'hmi.synoptic_mr_polfil_720s' series keyword. Note that they are large (1440 x 720), so you will probably want to downsample them to a smaller resolution to use them to calculate PFSS solutions.

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

```
import os

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

import pfsspy.utils
```

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.synoptic_mr_polfil_720s')
```

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

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

```
Results from 1 Provider:

177 Results from the JSOCClient:
Source: http://jsoc.stanford.edu

TELESCOP  INSTRUME WAVELNTH CAR_ROT
-------- --------- -------- -------
 SDO/HMI HMI_SIDE1   6173.0    2097
 SDO/HMI HMI_SIDE1   6173.0    2098
 SDO/HMI HMI_SIDE1   6173.0    2099
 SDO/HMI HMI_SIDE1   6173.0    2100
 SDO/HMI HMI_SIDE1   6173.0    2101
 SDO/HMI HMI_SIDE1   6173.0    2102
 SDO/HMI HMI_SIDE1   6173.0    2103
 SDO/HMI HMI_SIDE1   6173.0    2104
 SDO/HMI HMI_SIDE1   6173.0    2105
```

(continues on next page)

(continued from previous page)

```
 SDO/HMI HMI_SIDE1   6173.0    2106
     ...       ...      ...     ...
 SDO/HMI HMI_SIDE1   6173.0    2263
 SDO/HMI HMI_SIDE1   6173.0    2264
 SDO/HMI HMI_SIDE1   6173.0    2265
 SDO/HMI HMI_SIDE1   6173.0    2266
 SDO/HMI HMI_SIDE1   6173.0    2267
 SDO/HMI HMI_SIDE1   6173.0    2268
 SDO/HMI HMI_SIDE1   6173.0    2269
 SDO/HMI HMI_SIDE1   6173.0    2270
 SDO/HMI HMI_SIDE1   6173.0    2271
 SDO/HMI HMI_SIDE1   6173.0    2272
 SDO/HMI HMI_SIDE1   6173.0    2273
Length = 177 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(os.environ['JSOC_EMAIL']))
print(result)
```

```
Results from 1 Provider:

1 Results from the JSOCClient:
Source: http://jsoc.stanford.edu

TELESCOP  INSTRUME WAVELNTH CAR_ROT
--------- --------- -------- -------
 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)
```

```
Export request pending. [id=JSOC_20230824_1951_X_IN, status=2]
Waiting for 0 seconds...
1 URLs found for download. Full request totalling 4MB

Files Downloaded:   0%|          | 0/1 [00:00<?, ?file/s]
Files Downloaded: 100%|| 1/1 [00:00<00:00,  2.78file/s]
Files Downloaded: 100%|| 1/1 [00:00<00:00,  2.77file/s]
['/home/docs/sunpy/data/hmi.synoptic_mr_polfil_720s.2210.Mr_polfil.fits']
```

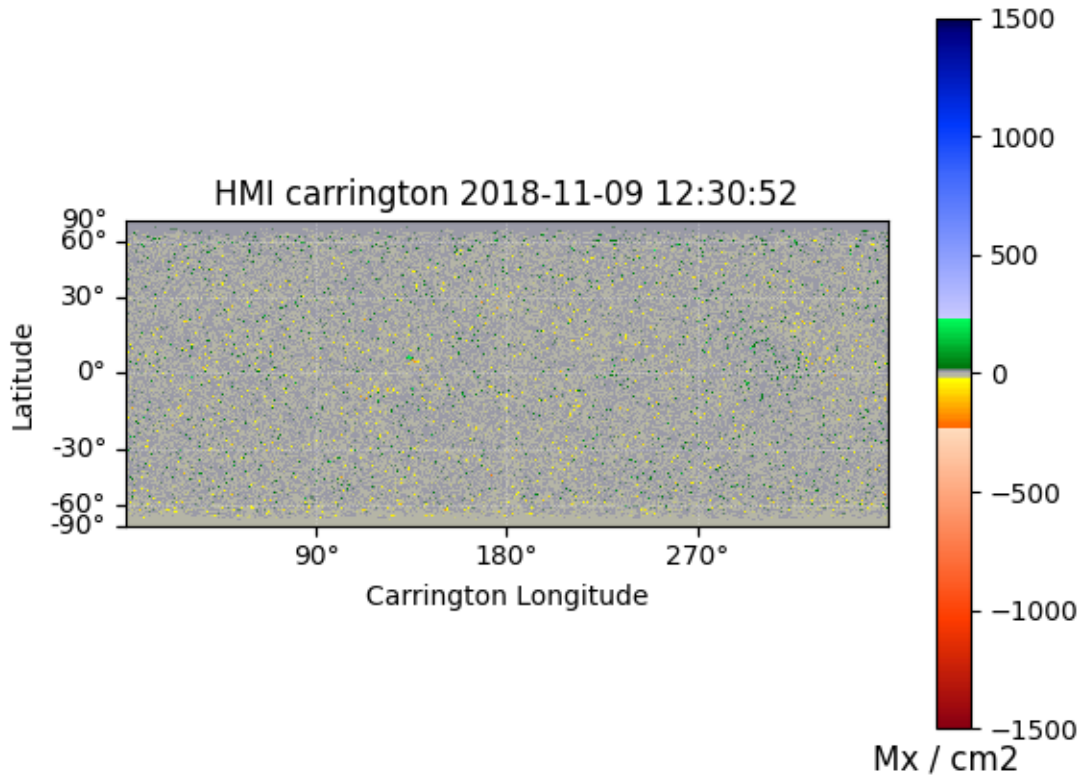
Read in a file. This will read in the first file downloaded to a sunpy Map object. Note that HMI maps have several bits of metadata that do not comply to the FITS standard, so we need to fix them first.

```
hmi_map = sunpy.map.Map(files[0])
pfsspy.utils.fix_hmi_meta(hmi_map)
```

(continues on next page)

(continued from previous page)

```
hmi_map.peek()
```




```
INFO: Missing metadata for solar radius: assuming the standard radius of the photosphere.
↪ [sunpy.map.mapbase]
INFO: Missing metadata for solar radius: assuming the standard radius of the photosphere.
↪ [sunpy.map.mapbase]
```

Total running time of the script: (0 minutes 14.769 seconds)

Parsing ADAPT Ensemble .fits files

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

```
import matplotlib.pyplot as plt
import sunpy.io
import sunpy.map
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 :

```
adapt_maps = 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)
for i, a_map in enumerate(adapt_maps):
    ax = fig.add_subplot(gs[i], projection=a_map)
    a_map.plot(axes=ax, cmap='bwr', vmin=-2, vmax=2,
               title=f"Realization {1+i:02d}")

plt.tight_layout(pad=5, h_pad=2)
plt.show()
```

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

Utilities

Useful code that doesn't involve doing a PFSS extrapolation.

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, a plate carée (CAR) projection, and need reprojecting.

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.

```
import matplotlib.pyplot as plt

from pfsspy import sample_data, utils
```

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]
```

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

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

pfsspy.adapt40311_03k012_202001010000_i00005600n1.fts.gz:   0%|          | 1.02k/3.11M␣
↪[00:00<06:35, 7.85kB/s]

pfsspy.adapt40311_03k012_202001010000_i00005600n1.fts.gz:   9%|         | 295k/3.11M␣
↪[00:00<00:01, 1.54MB/s]

pfsspy.adapt40311_03k012_202001010000_i00005600n1.fts.gz:  69%|   | 2.14M/3.11M [00:00
↪<00:00, 8.74MB/s]


Files Downloaded: 100%|| 1/1 [00:00<00:00,  2.02file/s]
Files Downloaded: 100%|| 1/1 [00:00<00:00,  2.01file/s]
```

Re-project into a CEA projection

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

```
INFO: Missing metadata for solar radius: assuming the standard radius of the photosphere.
↪ [sunpy.map.mapbase]
/home/docs/checkouts/readthedocs.org/user_builds/pfsspy/envs/latest/lib/python3.10/site-
↪packages/sunpy/map/mapbase.py:628: SunpyMetadataWarning: Missing metadata for␣
↪observer: assuming Earth-based observer.
For frame 'heliographic_stonyhurst' the following metadata is missing: dsun_obs,hglt_obs,
↪hgln_obs
For frame 'heliographic_carrington' the following metadata is missing: crlt_obs,dsun_obs,
↪crln_obs
```

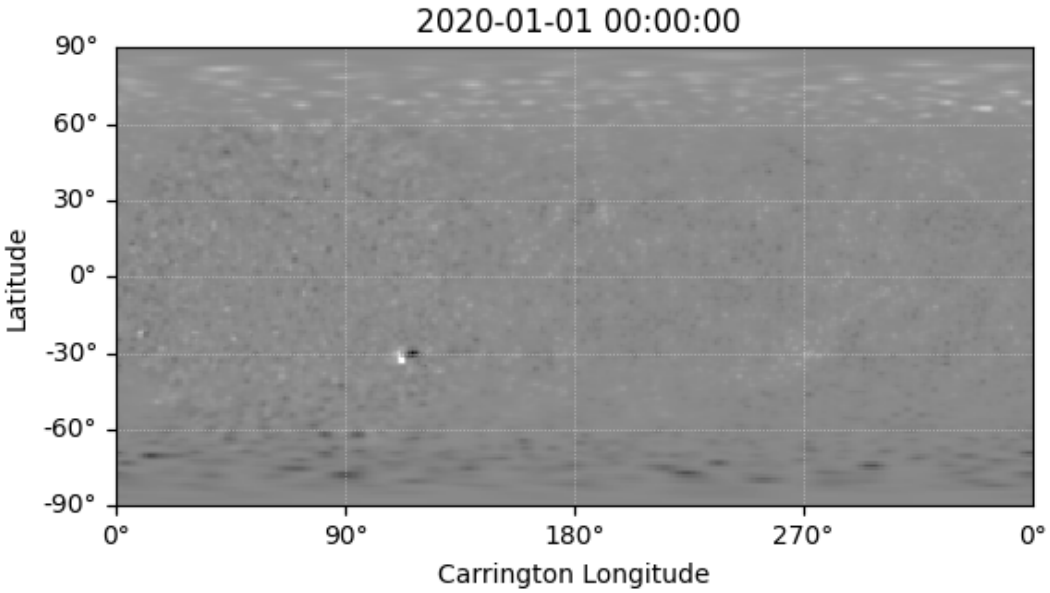
(continues on next page)

(continued from previous page)

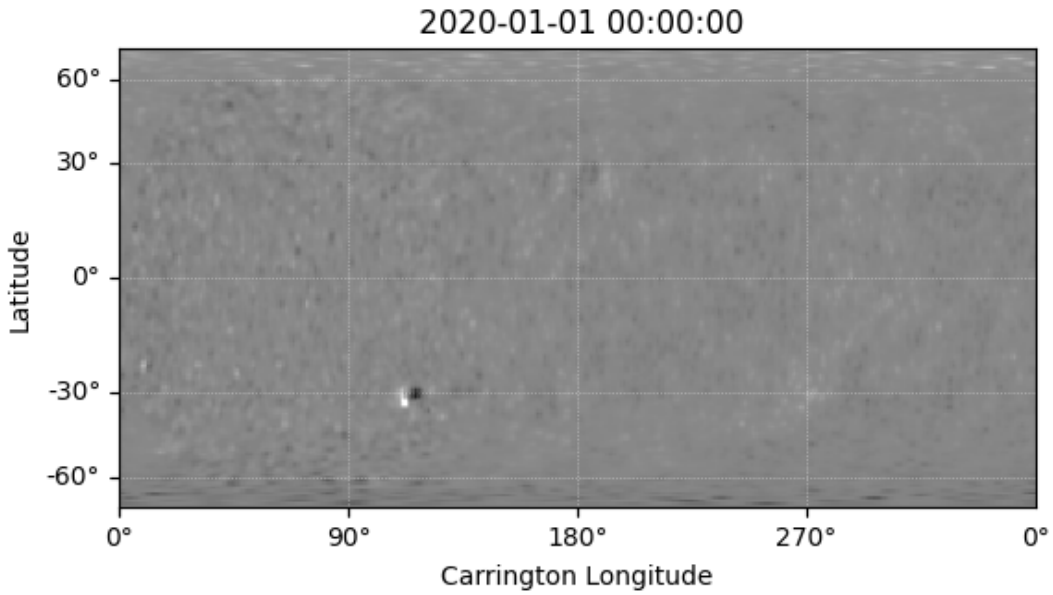
```
  obs_coord = self.observer_coordinate
```

Plot the original map and the reprojected map

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

plt.show()
```



-



-

Total running time of the script: (0 minutes 2.559 seconds)

Internals

Magnetic field grid

A plot of the grid corners which the magnetic field values are taken from 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 matplotlib.pyplot as plt
import numpy as np

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

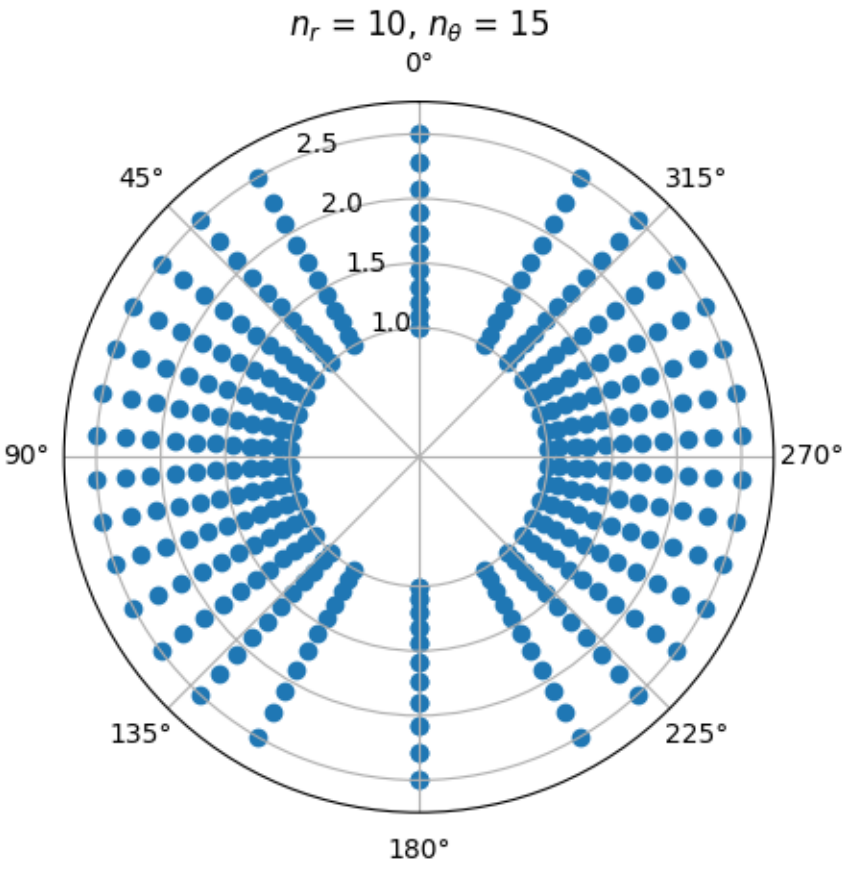
```
r_edges = np.exp(grid.rg)
theta_edges = np.arccos(grid.sg)
```

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 0.324 seconds)

Tracer performance

Comparing the performance of Python and FORTRAN tracers.

```
import timeit

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

import pfsspy
```

Create a dipole map

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

(continues on next page)

(continued from previous page)

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

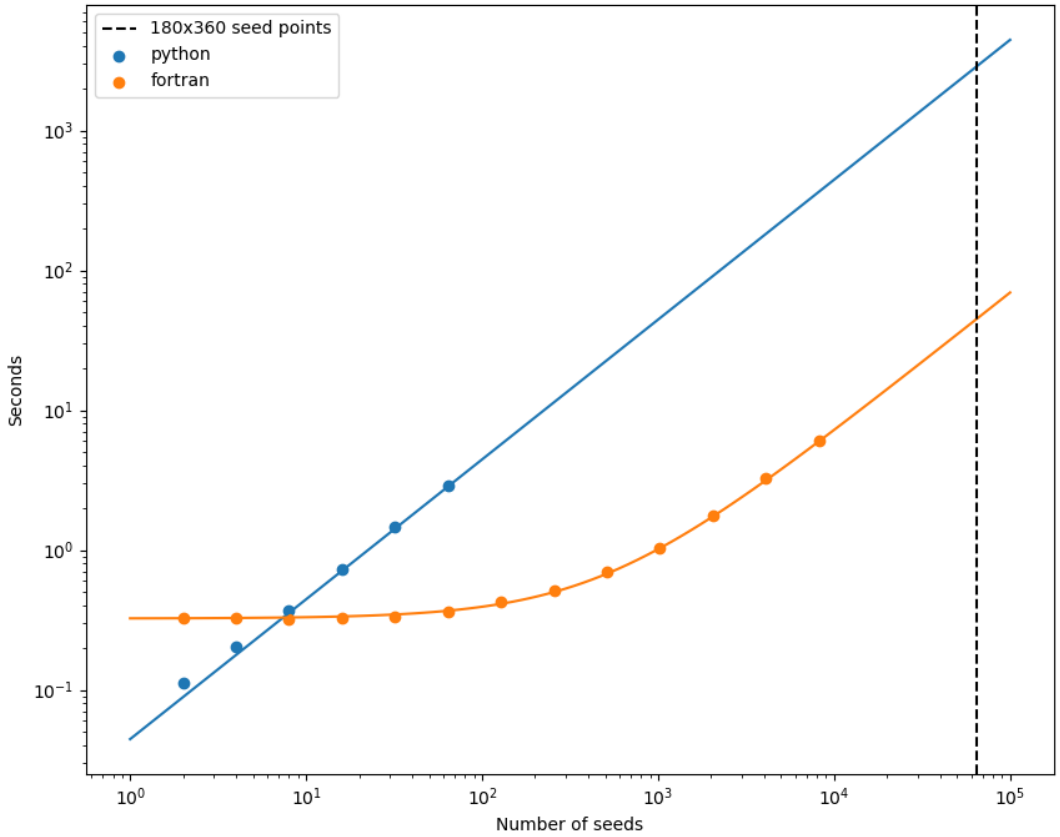
```
fig, ax = plt.subplots()
ax.scatter(nseeds[1:len(times[0])], times[0][1:], label='python')
ax.scatter(nseeds[1:], times[1][1:], label='fortran')

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))
```

(continues on next page)

(continued from previous page)

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

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

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)

Tests

Comparisons of the numerical output of pfsspy to analytic solutions.

Open flux and radial grid points (calcuations)

Comparing total unsigned flux to analytic solutions.

This script caclulates the ratio of numeric to analytic total unsigned open fluxes in PFSS solutions of spherical harmonics, as a function of the number of radial grid cells in the pfsspy grid.

```
import numpy as np
import pandas as pd
from tqdm import tqdm

from helpers import open_flux_analytic, open_flux_numeric, result_dir
```

Set the source surface height and range of radial grid points

```
zss = 2
nrhos = np.arange(10, 51, 2)
print(f'nrhos={nrhos}')
```

Loop through spherical harmonics and do the calculations. Only the ratio of fluxes between the analytic and numeric solutions is saved.

```
df = pd.DataFrame(index=nrhos, columns=[])

for l in range(1, 6):
    for m in range(-l, l+1):
        lm = str(l) + str(m)
        print(f"l={l}, m={m}")
        flux_analytic = open_flux_analytic(l, m, zss)
        flux_numeric = []
        for nrho in tqdm(nrhos):
            flux_numeric.append(open_flux_numeric(l, m, zss, nrho))
        flux_numeric = np.array(flux_numeric)
        flux_ratio = flux_numeric / flux_analytic
        df[lm] = flux_ratio
```

Save a copy of the data

```
df.to_csv(result_dir / 'open_flux_results.csv')
```

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

Open flux comparison (calculations)

Comparing total unsigned flux to analytic solutions.

This script calculates both analytic and numerical values of the total unsigned open flux within PFSS solutions of single spherical harmonics, and saves them to a .json file. This can be read in by `plot_open_flux_harmonics.py` to visualise the result.

```
import json
from collections import defaultdict

from helpers import open_flux_analytic, open_flux_numeric, result_dir
```

Set the source surface height, and the (l, m) values to investigate

```
zss = 2
nrho = 40

results = {'numeric': defaultdict(dict),
           'analytic': defaultdict(dict)}

for l in range(1, 6):
    for m in range(-l, l + 1):
        print(f"l={l}, m={m}")
        if -m in results['analytic'][l]:
            # Analytic flux for m = -m is the same
            flux_analytic = results['analytic'][l][-m]
        else:
            flux_analytic = open_flux_analytic(l, m, zss)

        results['analytic'][l][m] = float(flux_analytic)
        flux_numeric = open_flux_numeric(l, m, zss, nrho)
        results['numeric'][l][m] = float(flux_numeric)

# open file for writing, "w"
with open(result_dir / "open_flux_harmonics.json", "w") as f:
    # write json object to file
    f.write(json.dumps(results))
```

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

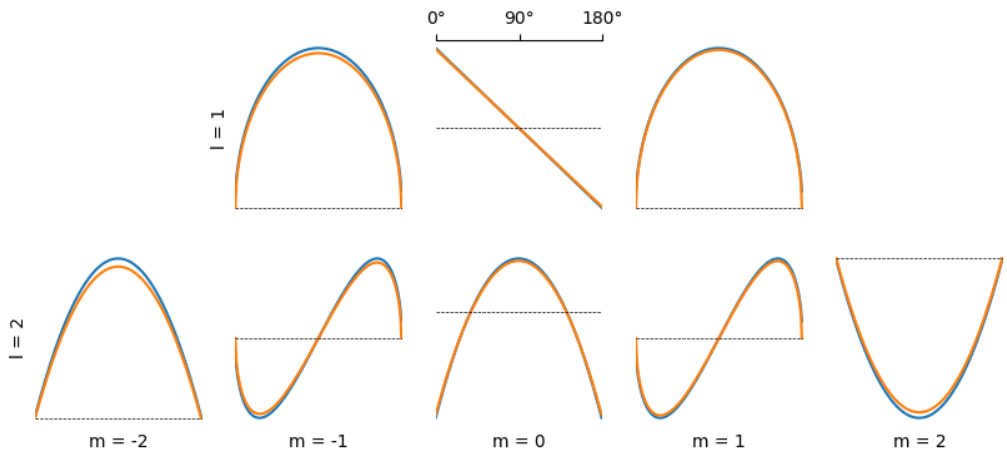
Spherical harmonic comparisons

Comparing analytical spherical harmonic solutions to PFSS output.

```
import matplotlib.pyplot as plt
import matplotlib.ticker as mticker

from helpers import LMAxes, brss_analytic, brss_pfsspy
```

Compare the the pfsspy solution to the analytic solutions. Cuts are taken on the source surface at a constant phi value to do a 1D comparison.

```
nphi = 360
ns = 180
rss = 2
nrho = 20

nl = 2
axs = LMAxes(nl=nl)

for l in range(1, nl+1):
    for m in range(-l, l+1):
        print(f'l={l}, m={m}')
        ax = axs[l, m]

        br_pfsspy = brss_pfsspy(nphi, ns, nrho, rss, l, m)
        br_actual = brss_analytic(nphi, ns, rss, l, m)

        ax.plot(br_pfsspy[:, 15], label='pfsspy')
        ax.plot(br_actual[:, 15], label='analytic')
        if l == 1 and m == 0:
            ax.xaxis.set_major_formatter(mticker.StrMethodFormatter('{x}°'))
            ax.xaxis.set_ticks([0, 90, 180])
            ax.xaxis.tick_top()
            ax.spines['top'].set_visible(True)
        ax.set_xlim(0, 180)
        ax.axhline(0, linestyle='--', linewidth=0.5, color='black')


plt.show()
```




```
l=1, m=-1
l=1, m=0
l=1, m=1
l=2, m=-2
l=2, m=-1
l=2, m=0
l=2, m=1
```

(continues on next page)

(continued from previous page)

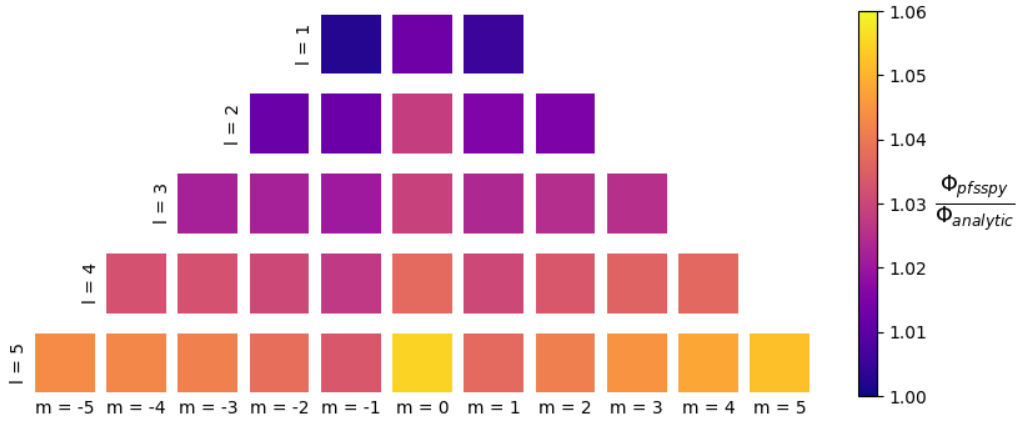
```
l=2, m=2
```

Total running time of the script: (0 minutes 38.603 seconds)

Open flux comparison

Comparing total unsigned flux to analytic solutions.

This script plots results generated by `open_flux_harmonics.py`, comparing analytic and numerical values of the total unsigned open flux within PFSS solutions of single spherical harmonics.




```
import json

import matplotlib.cm as cm
import matplotlib.colors as mcolor
import matplotlib.pyplot as plt

from helpers import LMAxes, result_dir

with open(result_dir / "open_flux_harmonics.json", "r") as f:
    results = json.load(f, parse_int=int)

axs = LMAxes(nl=5)
norm = mcolor.Normalize(vmin=1, vmax=1.06)
cmap = plt.get_cmap('plasma')

for lstr in results['analytic']:
    for mstr in results['analytic'][lstr]:
        l, m = int(lstr), int(mstr)

        ax = axs[l, m]
        ax.set_facecolor(cmap(norm(results['numeric'][lstr][mstr] /
                                   results['analytic'][lstr][mstr])))
        ax.set_aspect('equal')
```

(continues on next page)

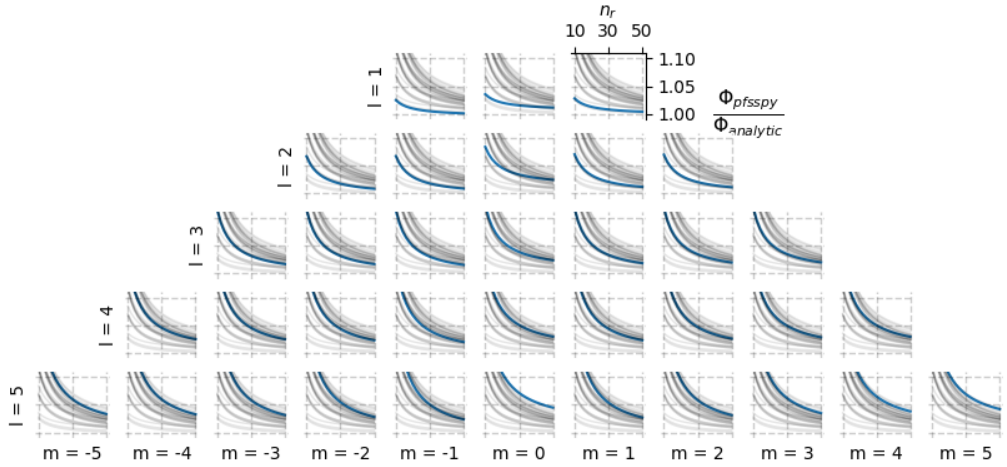
(continued from previous page)

```
cbar = axs.fig.colorbar(cm.ScalarMappable(norm=norm, cmap=cmap),
                        ax=axs.all_axs)
cbar.ax.set_ylabel(r'$\frac{\Phi_{pfsspy}}{\Phi_{analytic}}$', rotation=0,
                   size=18, labelpad=27, va='center')
plt.show()
```

Total running time of the script: (0 minutes 1.475 seconds)

Open flux and radial grid points

The script visualises results from `open_flux_harmonics.py`. It shows the ratio of numeric to analytic total unsigned open fluxes in PFSS solutions of spherical harmonics, as a function of the number of radial grid cells in the pfsspy grid.




```
import matplotlib.pyplot as plt
import matplotlib.ticker as mticker
import pandas as pd

from helpers import LMAxes, result_dir

df = pd.read_csv(result_dir / 'open_flux_results.csv', index_col=0)
axs = LMAxes(nl=5)

for lm in df.columns:
    l = int(lm[0])
    m = int(lm[1:])
    ax = axs[l, m]
    ax.plot(df.index, df[lm])

    for lm1 in df.columns:
        if lm1 != lm:
            ax.plot(df.index, df[lm1], linewidth=1, alpha=0.1, color='black')

    for x in [10, 30, 50]:
        ax.axvline(x, color='black', linestyle='--', linewidth=1, alpha=0.2)
    for y in [1, 1.05, 1.1]:
```

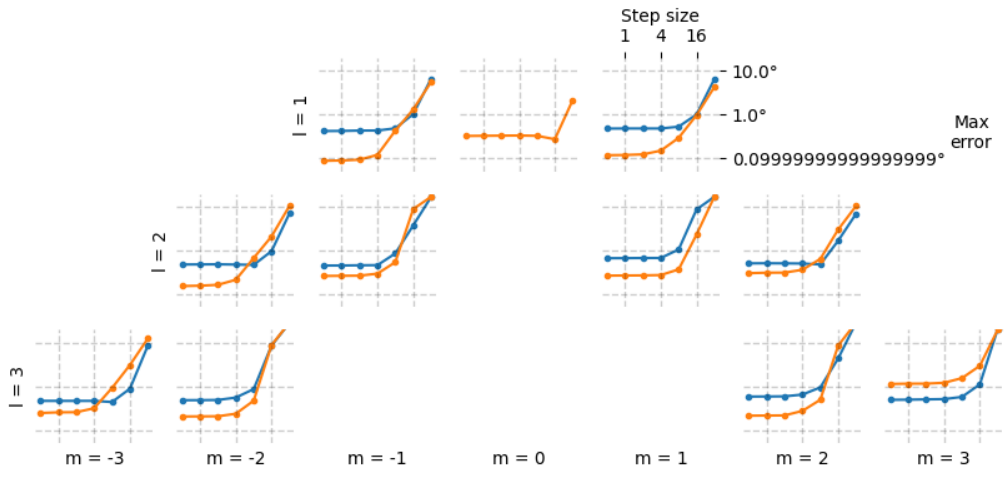
(continues on next page)

(continued from previous page)

```
        ax.axhline(y, color='black', linestyle='--', linewidth=1, alpha=0.2)
    ax.set_ylim(0.99, 1.11)

    if l == 1 and m == 1:
        ax.xaxis.set_ticks([10, 30, 50])
        ax.xaxis.tick_top()
        ax.set_xlabel(r'$n_{r}$')
        ax.xaxis.set_label_position('top')
        ax.xaxis.set_major_formatter(mticker.ScalarFormatter())

        ax.yaxis.set_ticks([1, 1.05, 1.1])
        ax.yaxis.tick_right()
        ax.set_ylabel(r'$\frac{\Phi_{pfsspy}}{\Phi_{analytic}}$',
                      rotation=0, labelpad=30, fontsize=16, loc='center')
        ax.yaxis.set_label_position('right')
        ax.yaxis.set_major_formatter(mticker.ScalarFormatter())

        ax.spines['top'].set_visible(True)
        ax.spines['right'].set_visible(True)

plt.show()
```

Total running time of the script: (0 minutes 4.753 seconds)

Tracer step size




```
1 -1
1 0
1 1
2 -2
2 -1
 Files not found for l=2, m=0
2 1
2 2
```

(continues on next page)

(continued from previous page)

```
3 -3
3 -2
 Files not found for l=3, m=-1
 Files not found for l=3, m=0
 Files not found for l=3, m=1
3 2
3 3
```

```
import matplotlib.pyplot as plt
import matplotlib.ticker as mticker
import pandas as pd

from helpers import LMAxes, result_dir

nl = 3

axs = LMAxes(nl=nl)

for l in range(1, nl+1):
    for m in range(-l, l+1):
        ax = axs[l, m]
        try:
            dphis = pd.read_csv(result_dir / f'flines/dphis_{l}{m}.csv',
                                header=None, index_col=0)
            dthetas = pd.read_csv(result_dir / f'flines/dthetas_{l}{m}.csv',
                                  header=None, index_col=0)
            print(l, m)
        except FileNotFoundError:
            print(f' Files not found for l={l}, m={m}')
            continue

        for data in [dphis, dthetas]:
            ax.plot(data.index, data.values, marker='.')

        ax.set_xscale('log')
        ax.set_yscale('log')
        ax.set_ylim(0.5e-1, 2e1)
        for x in [1, 4, 16]:
            ax.axvline(x, color='k', linewidth=1, linestyle='--', alpha=0.2)
        for y in [1e-1, 1, 1e1]:
            ax.axhline(y, color='k', linewidth=1, linestyle='--', alpha=0.2)

        if l == 1 and m == 1:
            ax.xaxis.tick_top()
            ax.yaxis.tick_right()
            ax.xaxis.set_ticks([1, 4, 16])
            ax.xaxis.set_major_formatter(mticker.ScalarFormatter())
```

(continues on next page)

(continued from previous page)

```
            ax.yaxis.set_major_formatter(mticker.StrMethodFormatter('{x}°'))
            ax.xaxis.set_ticks([], minor=True)
            ax.yaxis.set_ticks([], minor=True)
            ax.set_xlabel('Step size')
            ax.set_ylabel('Max\nerror', rotation=0, labelpad=15, va='center')
            ax.xaxis.set_label_position('top')
            ax.yaxis.set_label_position('right')
        else:
            ax.yaxis.set_major_formatter(mticker.NullFormatter())
            for minor in [True, False]:
                ax.xaxis.set_ticks([], minor=minor)
                ax.yaxis.set_ticks([], minor=minor)


plt.show()
```

Total running time of the script: (0 minutes 1.207 seconds)

Tracer step size (calculations)

```
import astropy.constants as const
import astropy.units as u
import numpy as np
import pandas as pd
from astropy.coordinates import SkyCoord

from pfsspy import tracing

from helpers import (
    pffspy_output,
    phi_fline_coords,
    result_dir,
    theta_fline_coords,
)

l = 3
m = 3
nphi = 360
ns = 180
nr = 40
rss = 2
```

Calculate PFSS solution

```
pfsspy_out = pffspy_output(nphi, ns, nr, rss, l, m)
```

Trace an array of field lines from the source surface down to the solar surface

```
n = 90
# Create 1D theta, phi arrays
phi = np.linspace(0, 360, n * 2)
```

(continues on next page)

(continued from previous page)

```
phi = phi[:-1] + np.diff(phi) / 2
theta = np.arcsin(np.linspace(-0.98, 0.98, n, endpoint=False) + 1/n)
# Mesh into 2D arrays
theta, phi = np.meshgrid(theta, phi, indexing='ij')
theta, phi = theta * u.rad, phi * u.deg
rss = rss * const.R_sun
seeds = SkyCoord(radius=rss, lat=theta.ravel(), lon=phi.ravel(),
                 frame=pfsspy_out.coordinate_frame)

step_sizes = [32, 16, 8, 4, 2, 1, 0.5]
dthetas = []
dphis = []
for step_size in step_sizes:
    print(f'Tracing {step_size}...')
    # Trace
    tracer = tracing.FortranTracer(step_size=step_size)
    flines = tracer.trace(seeds, pfsspy_out)
    # Set a mask of open field lines
    mask = flines.connectivities.astype(bool).reshape(theta.shape)

    # Get solar surface latitude
    phi_solar = np.ones_like(phi) * np.nan
    phi_solar[mask] = flines.open_field_lines.solar_feet.lon
    theta_solar = np.ones_like(theta) * np.nan
    theta_solar[mask] = flines.open_field_lines.solar_feet.lat
    r_out = np.ones_like(theta.value) * const.R_sun * np.nan
    r_out[mask] = flines.open_field_lines.solar_feet.radius

    # Calculate analytical solution
    theta_analytic = theta_fline_coords(r_out, rss, l, m, theta)
    dtheta = (theta_solar - theta_analytic).to_value(u.deg)
    phi_analytic = phi_fline_coords(r_out, rss, l, m, theta, phi)
    dphi = (phi_solar - phi_analytic).to_value(u.deg)
    # Wrap phi values
    dphi[dphi > 180] -= 360
    dphi[dphi < -180] += 360

    dthetas.append(dtheta.ravel())
    dphis.append(dphi.ravel())
```

Save results. This saves the maximum error in both phi and theta as a function of thet tracer step size.

```
dthetas = pd.DataFrame(data=np.array(dthetas), index=step_sizes)
dthetas = dthetas.mask(np.abs(dthetas) > 30).max(axis=1)

dphis = pd.DataFrame(data=np.array(dphis), index=step_sizes)
dphis = dphis.mask(np.abs(dphis) > 30).max(axis=1)

dthetas.to_csv(result_dir / f'flines/dthetas_{l}{m}.csv', header=False)
dphis.to_csv(result_dir / f'flines/dphis_{l}{m}.csv', header=False)
```

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

Field line error map

This script produces a map of errors between analytic field line equations and field lines numerically traced by pfsspy.

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

from pfsspy import tracing

from helpers import pffspy_output, phi_fline_coords, theta_fline_coords

quantity_support()

l = 3
m = 3
nphi = 360
ns = 180
nr = 40
rss = 2
```

Calculate PFSS solution

```
pfsspy_out = pffspy_output(nphi, ns, nr, rss, l, m)

rss = rss * const.R_sun
```

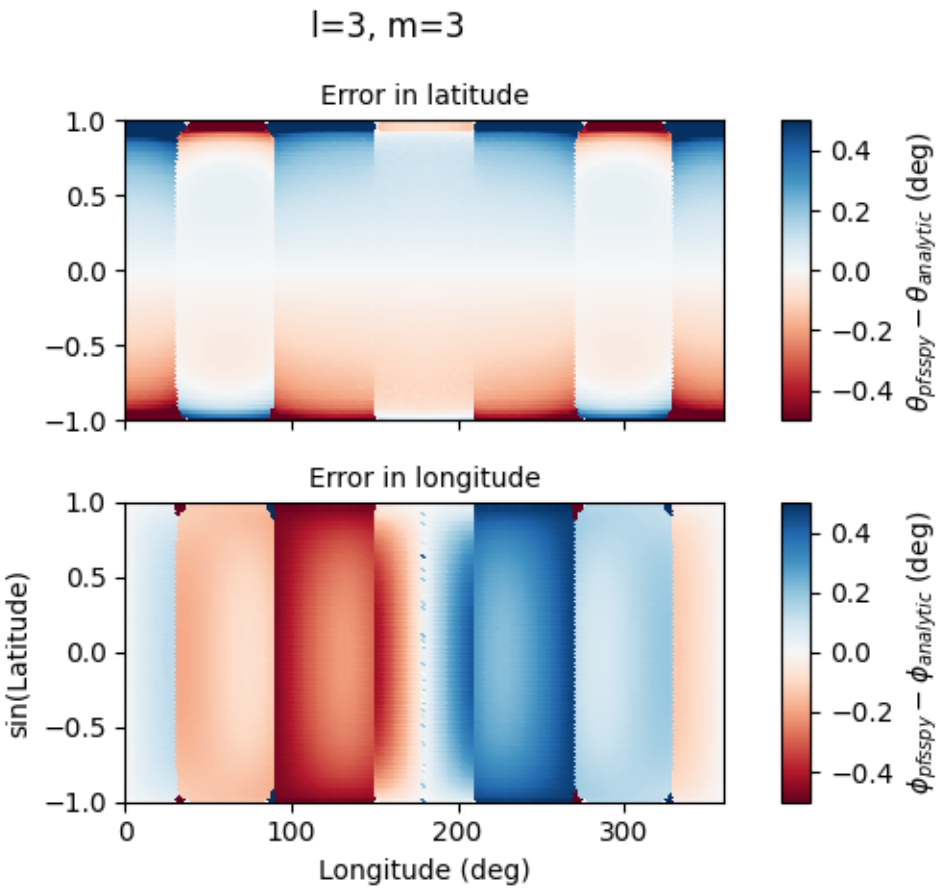
Trace field lines

```
n = 90
# Create 1D theta, phi arrays
phi = np.linspace(0, 360, n * 2)
phi = phi[:-1] + np.diff(phi) / 2
theta = np.arcsin(np.linspace(-0.98, 0.98, n, endpoint=False) + 1/n)
# Mesh into 2D arrays
theta, phi = np.meshgrid(theta, phi, indexing='ij')
theta, phi = theta * u.rad, phi * u.deg
seeds = SkyCoord(radius=rss, lat=theta.ravel(), lon=phi.ravel(),
                 frame=pfsspy_out.coordinate_frame)

step_size = 1
dthetas = []
print(f'Tracing {step_size}...')
# Trace
tracer = tracing.FortranTracer(step_size=step_size)
flines = tracer.trace(seeds, pfsspy_out)
# Set a mask of open field lines
mask = flines.connectivities.astype(bool).reshape(theta.shape)

r_out = np.ones_like(theta.value) * const.R_sun * np.nan
r_out[mask] = flines.open_field_lines.solar_feet.radius
```

(continues on next page)

(continued from previous page)

```
# longitude
phi_solar = np.ones_like(phi) * np.nan
phi_analytic = np.ones_like(phi) * np.nan
phi_solar[mask] = flines.open_field_lines.solar_feet.lon
try:
    phi_analytic = phi_fline_coords(r_out, rss, l, m, theta, phi)
except KeyError:
    # If there's no g_lm entry
    print(f'No g_lm entry for l={l}, m={m}')
dphi = phi_solar - phi_analytic

theta_solar = np.ones_like(theta) * np.nan
theta_solar[mask] = flines.open_field_lines.solar_feet.lat
theta_analytic = theta_fline_coords(r_out, rss, l, m, theta)
dtheta = theta_solar - theta_analytic

fig, axs = plt.subplots(nrows=2, sharex=True, sharey=True)


def plot_map(field, ax, label, title):
    kwargs = dict(cmap='RdBu', vmin=-0.5, vmax=0.5, shading='nearest', edgecolors='face')
    im = ax.pcolormesh(phi.to_value(u.deg), np.sin(theta).value,
                       field, **kwargs)
    ax.set_aspect(360 / 4)
    fig.colorbar(im, aspect=10, ax=ax,
                 label=label)
    ax.set_title(title, size=10)


plot_map(dtheta.to_value(u.deg), axs[0],
         r'$\theta_{pfsspy} - \theta_{analytic}$ (deg)',
         'Error in latitude')
plot_map(dphi.to_value(u.deg), axs[1],
         r'$\phi_{pfsspy} - \phi_{analytic}$ (deg)',
         'Error in longitude')

ax = axs[1]
ax.set_xlim(0, 360)
ax.set_ylim(-1, 1)
ax.set_xlabel('Longitude (deg)')
ax.set_ylabel('sin(Latitude)')

fig.suptitle(f'l={l}, m={m}')
fig.tight_layout()

plt.show()
```




```
Tracing 1...
```

Total running time of the script: (0 minutes 17.988 seconds)

1.3 API reference

1.3.1 pfsspy Package

Functions

pfss(input)	Compute PFSS model.

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.

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.

Notes

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.

Attributes Summary

grid	*Grid* that the PFSS solution for this input is calculated on.
map	sunpy.map.GenericMap representation of the input.

Attributes Documentation

grid

Grid that the PFSS solution for this input is calculated on.

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	Coordinate frame of the input map.
dtime	Date and time of the input map.
source_surface_br	Radial magnetic field component on the source surface.
source_surface_pils	Coordinates of the polarity inversion lines on the source surface.

Methods Summary

get_bvec(coords[, out_type])	Interpolate magnetic vectors at arbitrary coordinates.
trace(tracer, seeds)	Trace magnetic field lines.

Attributes Documentation

bc

B on the centres of the cell faces.

Returns

- **br** (*astropy.units.Quantity*) – A `nphi, ns, nrho + 1` shaped Quantity.
- **btheta** (*astropy.units.Quantity*) – A `nphi, ns + 1, nrho` shaped Quantity.
- **bphi** (*astropy.units.Quantity*) – A `nphi + 1, ns, nrho` shaped Quantity.

Notes

The three components are not co-located at the same locations. Component `n` is located on cell faces of constant `n`, e.g. the `r` component is located on the cell faces at constant `r` values.

bg

B as a (weighted) averaged on grid points.

Returns

A `(nphi + 1, ns + 1, nrho + 1, 3)` shaped Quantity. The last index gives the corodinate axis, 0 for Bphi, 1 for Bs, 2 for Brho. Because the phi dimension is periodic, `bg[0, :, :] == bg[-1, :, :]`.

Return type

astropy.units.Quantity

bunit

Unit of the input map data.

If no metadata is available, returns dimensionless units.

coordinate_frame

Coordinate frame of the input map.

Notes

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

dtime

Date and time of the input map.

source_surface_br

Radial magnetic field component on the source surface.

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 SkyCoord, as in general there may be more than one polarity inversion line.

Methods Documentation

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

Interpolate magnetic vectors at arbitrary coordinates.

Parameters

- **coords** (astropy.coordinates.SkyCoord) – An arbitary point or set of points (length N >= 1) in the PFSS model domain (1Rs < r < Rss).
- **out_type** (*str*) – 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 – (N, 3) shaped Quantity of magnetic field vectors.

Return type
astropy.units.Quantity

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)

trace(*tracer*, *seeds*)

Trace magnetic field lines.

Parameters

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

Class Inheritance Diagram

1.3.2 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

1.3.3 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: *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: *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

1.3.4 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='auto'*, *step_size=1*)

Bases: *Tracer*

Tracer using Fortran code.

Parameters

- **max_steps** (*str*, *int*) – Maximum number of steps each streamline can take before stopping. This controls the total amount of memory allocated for all the streamlines.

  If `'auto'` (the default) this is set to $4n_r/ds$, where n_r is the number of radial grid points and ds is the specified `step_size`.

- **step_size** (*float*) – Step size as a fraction of numerical grid cell size at the equator. Must be less than the number of radial coordinate cells.

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: *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

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

1.3.5 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.
fix_hmi_meta(hmi_map)	Fix non-compliant FITS metadata in HMI maps.
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`
roll_map(m[, lh_edge_lon, method])	Roll an input synoptic map so that it's left edge corresponds to a specific Carrington longitude.

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*, *, *map_center_longitude=<Quantity 0. deg>*)

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.
- **map_center_longitude** (*astropy.units.Quantity*) – Change the world coordinate longitude of the central image pixel to allow for different roll angles of the Carrington map. Default to 0 deg. Must be supplied with units of `astropy.units.deg`

References

fix_hmi_meta

pfsspy.utils.**fix_hmi_meta**(*hmi_map*)

Fix non-compliant FITS metadata in HMI maps.

This function:

- Corrects CUNIT2 from 'Sine Latitude' to 'deg'
- Corrects CDELT1 and CDELT2 for a CEA projection

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

- Populates the DATE-OBS keyword from the T_OBS keyword
- Sets the observer coordinate to the Earth

Notes

If you have sunpy > 2.1 installed, this function is not needed as sunpy will automatically make these fixes.

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.

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

roll_map

pfsspy.utils.**roll_map**(*m*, *lh_edge_lon: Unit("deg") = <Quantity 0. deg>*, *method='interp'*)

Roll an input synoptic map so that it's left edge corresponds to a specific Carrington longitude.

Roll is performed by changing the FITS header parameter "CRVAL1" to the new value of the reference pixel (FITS header parameter CRPIX1) corresponding to aligning the left hand edge of the map with lh_edge_lon. The altered header is provided to reproject to produce the new map.

Parameters

- **m** (*sunpy.map.GenericMap*) – Input map
- **lh_edge_lon** (*float*) – Desired Carrington longitude (degrees) for left hand edge of map. Default is 0.0 which results in a map with the edges at 0/360 degrees Carrington longitude. Input value must be in the range [0,360]
- **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 CRVAL1 which encodes the longitude of the reference pixel in the image, and which is updated to produce the correct roll.

Return type
sunpy.map.GenericMap

See also:

`reproject`

1.3.6 pfsspy.analytic Module

Analytic inputs and solutions to the PFSS equations.

This sub-module contains functions to generate solutions to the PFSS equations in the case where the input field is a single spherical harmonic, specified with the spherical harmonic numbers `l, m`.

All angular quantities must be passed as astropy quantities. All radial quantities are passed normalised to the source surface radius, and therefore can be passed as normal scalar values.

Angular definitions

- `theta` is the polar angle, in the range $0, \pi$ (ie. the co-latitude).
- `phi` is the azimuthal angle, in the range $0, 2\pi$.

Using this module requires `sympy` to be installed.

Functions

Bphi(l, m, zss)	Analytic phi component of magnetic field on the source surface.
Br(l, m, zss)	Analytic radial component of magnetic field on the source surface.
Btheta(l, m, zss)	Analytic theta component of magnetic field on the source surface.

Bphi

pfsspy.analytic.**Bphi**(*l*, *m*, *zss*)

Analytic phi component of magnetic field on the source surface.

Parameters

- **l** (*int*) – Spherical harmonic numbers.
- **m** (*int*) – Spherical harmonic numbers.
- **zss** (*float*) – Source surface radius (as a fraction of the solar radius).

Returns
Has the signature `Bphi(z, theta, phi)`.

Return type
function

Br

pfsspy.analytic.**Br**(*l*, *m*, *zss*)

Analytic radial component of magnetic field on the source surface.

Parameters

- **l** (*int*) – Spherical harmonic numbers.
- **m** (*int*) – Spherical harmonic numbers.
- **zss** (*float*) – Source surface radius (as a fraction of the solar radius).

Returns
Has the signature `Br(z, theta, phi)`.

Return type
function

Btheta

pfsspy.analytic.**Btheta**(*l*, *m*, *zss*)

Analytic theta component of magnetic field on the source surface.

Parameters

- **l** (*int*) – Spherical harmonic numbers.
- **m** (*int*) – Spherical harmonic numbers.
- **zss** (*float*) – Source surface radius (as a fraction of the solar radius).

Returns
Has the signature `Btheta(z, theta, phi)`.

Return type
function

1.4 Improving performance

1.4.1 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.

1.4.2 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.

1.5 Changelog

1.5.1 1.2.0

- Allow the center of map coordinates to be specified in *pfsspy.utils.carr_cea_wcs_header*.
- Fixed a deprecation warning emitted with sunpy 4.1.
- Bumped the minimum Python version to 3.9, and added explicit support for Python 3.11.
- Bumped the minimum version of sunpy to 4.0.

1.5.2 1.1.2

- Added project status documentation.
- Bumped the minimum version of astropy to 5.0.
- Fixed ADAPT map reading with sunpy >= 4.0.

1.5.3 1.1.1

Fixed imports so pfsspy does not depend on `sympy` as a runtime dependency. (`sympy` is still needed for the `analytic` module however).

1.5.4 1.1.0

New requirements

pfsspy now depends on Python >= 3.8, and is officially supported with Python 3.10

New examples

A host of new examples comparing pfsspy results to analytic solutions have been added to the example gallery.

Bug fixes

- Updated the sunpy package requirement to include all packages needed to use sunpy maps.
- Any traced field line points that are out of bounds in latitude (ie. have a latitude > 90 deg) are now filtered out. This was previously only an issue for very low tracing step sizes.

1.5.5 1.0.1

Bug fixes

- Fixed compatibility of map validity checks with sunpy 3.1.
- Updated this changelog to make it clear that pfsspy 1.0.0 depends on sunpy >= 3.0.

1.5.6 1.0.0

New requirements

pfsspy now depends on python >= 3.7, sunpy >=3, and now does *not* depend on Matplotlib.

New features

- The `max_steps` argument to `pfsspy.tracers.FortranTracer` now defaults to `'auto'` and automatically sets the maximum number of steps to four times the number of steps that are needed to span radially from the solar to source surface. `max_steps` can still be manually specified as a number if more or less steps are desired.
- *FieldLines* now has a `__len__` method, meaning one can now do `n_field_lines = len(my_field_lines)`.
- Added *pfsspy.utils.roll_map()* to roll a map in the longitude direction. This is particularly helpful to modify GONG maps so they have a common longitude axis.
- Added the *pfsspy.analytic* sub-module that provides functions to sample analytic solutions to the PFSS equations.

Bug fixes

- *pfsspy.utils.carr_cea_wcs_header()* now works with versions of sunpy >= 2.0.
- GONG synoptic maps now automatically have their observer information corrected (by assuming an Earth observer) when loaded by `sunpy.map.Map`.
- The plot settings of input maps are no longer modified in place.

Breaking changes

- The interpretation of the `step_size` to `pfsspy.tracers.FortranTracer` has been corrected so that it is the step size relative to the radial cell size. A step size of 0.01 specified in pfsspy<1.0 is approximately equivalent to a step size of 1 in pfsspy 1.0, so you will need to adjust any custom step sizes accordingly.
- Any points on field lines that are out of bounds (ie. below the solar surface or above the source surface) are now removed by the *FortranTracer*.
- *pfsspy.pfss()* no longer warns if the mean of the input data is non-zero, and silently ignores the monopole component.

Removals

- Saving and load PFSS solutions is no longer possible. This was poorly tested, and possibly broken. If you have interest in saving and loading being added as a new feature to pfsspy, please open a new issue at https://github.com/dstansby/pfsspy/issues.

1.5.7 0.6.6

Two bugs have been fixed in *pfsspy.utils.carr_cea_wcs_header*:

- The reference pixel was previously one pixel too large in both longitude and latitude.
- The longitude coordinate was previously erroneously translated by one degree.

Both of these are now fixed.

1.5.8 0.6.5

This release improves documentation and handling of HMI maps. In particular:

- The HMI map downloading example has been updated to use the polar filled data product, which does not have any data missing at the poles.
- *pfsspy.utils.fix_hmi_meta()* has been added to fix metadata issues in HMI maps. This modifies the metadata of a HMI map to make it FITS compliant, allowing it to be used with pfsspy.

1.5.9 0.6.4

This release adds citation information to the documentation.

1.5.10 0.6.3

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

1.5.11 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.

1.5.12 0.6.1

Bug fixes

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

1.5.13 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.

1.5.14 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.

1.5.15 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.

1.5.16 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.

1.5.17 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.

1.5.18 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.

1.5.19 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.

1.5.20 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.

1.5.21 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.

1.5.22 0.3.2

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

1.5.23 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.

1.5.24 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.

1.5.25 0.2.0

- *pfsspy.Input* and *pfsspy.Output* now take the optional keyword argument *dtime*, which stores the date-time 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*

1.5.26 0.1.5

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

1.5.27 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.

1.5.28 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.

1.6 History, status, and future

The original PFSS implementation in Python was written by Anthony Yeates. I (David Stansby) then took this and added tests, documentation, and integrated it with the SunPy project ecosystem to create the pfsspy package. Most of the package was written by myself, with some community contributions.

I am the sole maintainer of the package, which is currently feature complete (providing a spherical PFSS solver in Python that integrates well with the SunPy package ecosystem).

Going forward the only changes to the package will be:

- Fixing identified bugs
- Adding tightly scoped new features to improve usability or integrate better with other SunPy packages.
- Improving the documentation.
- Retaining compatibility with future versions of the package dependencies (e.g. sunpy, Matplotlib...)

I intend to remain the sole maintainer of the package, but since I'm not currently active in solar physics research don't have lots of time to work on pfsspy. As a priority I will maintain the issue list with well scoped pieces of work that others can contribute to fixing.

1.6.1 Community contributions

The pfsspy issue tracker maintains a list of items that need working on. Please open a new issue if you find a problem or want to see a new feature added! If you want to fix any of the issues yourself, please open a pull request. I will try my best to review and merge pull requests, but **please keep them as small as possible** to make my life easier! General guidelines for contributing to open source can be found in the sunpy developers guide

1.7 Numerical methods

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

1.8 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.

1.8.1 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

TWO

CITING

If you use pfsspy in work that results in publication, please cite the Journal of Open Source Software paper at https://doi.org/10.21105/joss.02732. A ready made bibtex entry is

```
@article{Stansby2020,
  doi = {10.21105/joss.02732},
  url = {https://doi.org/10.21105/joss.02732},
  year = {2020},
  publisher = {The Open Journal},
  volume = {5},
  number = {54},
  pages = {2732},
  author = {David Stansby and Anthony Yeates and Samuel T. Badman},
  title = {pfsspy: A Python package for potential field source surface modelling},
  journal = {Journal of Open Source Software}
}
```

PYTHON MODULE INDEX

p

pfsspy, 50
pfsspy.analytic, 66
pfsspy.fieldline, 57
pfsspy.grid, 55
pfsspy.tracing, 60
pfsspy.utils, 63

INDEX

B

b_along_fline (*pfsspy.fieldline.FieldLine attribute*), 58
bc (*pfsspy.Output attribute*), 53
bg (*pfsspy.Output attribute*), 53
Bphi() (*in module pfsspy.analytic*), 67
Br() (*in module pfsspy.analytic*), 67
Btheta() (*in module pfsspy.analytic*), 68
bunit (*pfsspy.Output attribute*), 53

C

car_to_cea() (*in module pfsspy.utils*), 64
carr_cea_wcs_header() (*in module pfsspy.utils*), 64
cartesian_to_coordinate() (*pfsspy.tracing.Tracer static method*), 63
closed_field_lines (*pfsspy.fieldline.FieldLines attribute*), 59
ClosedFieldLines (*class in pfsspy.fieldline*), 57
connectivities (*pfsspy.fieldline.FieldLines attribute*), 59
coordinate_frame (*pfsspy.Output attribute*), 53
coords (*pfsspy.fieldline.FieldLine attribute*), 58
coords_to_xyz() (*pfsspy.tracing.Tracer static method*), 63

D

dp (*pfsspy.grid.Grid attribute*), 56
dr (*pfsspy.grid.Grid attribute*), 56
ds (*pfsspy.grid.Grid attribute*), 56
dtime (*pfsspy.Output attribute*), 54

E

expansion_factor (*pfsspy.fieldline.FieldLine attribute*), 58
expansion_factors (*pfsspy.fieldline.FieldLines attribute*), 59

F

FieldLine (*class in pfsspy.fieldline*), 57
FieldLines (*class in pfsspy.fieldline*), 59
fix_hmi_meta() (*in module pfsspy.utils*), 64
FortranTracer (*class in pfsspy.tracing*), 60

G

get_bvec() (*pfsspy.Output method*), 54
Grid (*class in pfsspy.grid*), 55
grid (*pfsspy.Input attribute*), 52

I

Input (*class in pfsspy*), 51
is_car_map() (*in module pfsspy.utils*), 65
is_cea_map() (*in module pfsspy.utils*), 65
is_full_sun_synoptic_map() (*in module pfsspy.utils*), 65
is_open (*pfsspy.fieldline.FieldLine attribute*), 58

L

load_adapt() (*in module pfsspy.utils*), 65

M

map (*pfsspy.Input attribute*), 52
module
    pfsspy, 50
    pfsspy.analytic, 66
    pfsspy.fieldline, 57
    pfsspy.grid, 55
    pfsspy.tracing, 60
    pfsspy.utils, 63

O

open_field_lines (*pfsspy.fieldline.FieldLines attribute*), 59
OpenFieldLines (*class in pfsspy.fieldline*), 60
Output (*class in pfsspy*), 52

P

pc (*pfsspy.grid.Grid attribute*), 56
pfss() (*in module pfsspy*), 51
pfsspy
    module, 50
pfsspy.analytic
    module, 66
pfsspy.fieldline
    module, 57

pfsspy.grid
    module, 55
pfsspy.tracing
    module, 60
pfsspy.utils
    module, 63
pg (*pfsspy.grid.Grid attribute*), 56
polarities (*pfsspy.fieldline.FieldLines attribute*), 59
polarity (*pfsspy.fieldline.FieldLine attribute*), 58
PythonTracer (*class in pfsspy.tracing*), 61

R

rc (*pfsspy.grid.Grid attribute*), 56
rg (*pfsspy.grid.Grid attribute*), 56
roll_map() (*in module pfsspy.utils*), 66

S

sc (*pfsspy.grid.Grid attribute*), 56
sg (*pfsspy.grid.Grid attribute*), 56
solar_feet (*pfsspy.fieldline.OpenFieldLines attribute*), 60
solar_footpoint (*pfsspy.fieldline.FieldLine attribute*), 58
source_surface_br (*pfsspy.Output attribute*), 54
source_surface_feet (*pfsspy.fieldline.OpenFieldLines attribute*), 60
source_surface_footpoint (*pfsspy.fieldline.FieldLine attribute*), 58
source_surface_pils (*pfsspy.Output attribute*), 54

T

trace() (*pfsspy.Output method*), 55
trace() (*pfsspy.tracing.FortranTracer method*), 61
trace() (*pfsspy.tracing.PythonTracer method*), 62
trace() (*pfsspy.tracing.Tracer method*), 63
Tracer (*class in pfsspy.tracing*), 62

V

validate_seeds() (*pfsspy.tracing.Tracer static method*), 63
vector_grid() (*pfsspy.tracing.FortranTracer static method*), 61