pfsspy Documentation

pfsspy contributors

May 27, 2020

CONTENTS

pfsspy is a python package for carrying out Potential Field Source Surface modelling. For more information on the actually PFSS calculation see this document.

Note: pfsspy is a very new package, so elements of the API are liable to change with the first few releases. If you find any bugs or have any suggestions for improvement, please raise an issue here: https://github.com/dstansby/pfsspy/issues

---

pfsspy can be installed from PyPi using

```
pip install pfsspy
```

CHAPTER

ONE

IMPROVING PERFORMANCE

1.1 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.

CHAPTER

TWO

CITING

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

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

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

CHAPTER

THREE

CODE REFERENCE

For the main user-facing code and a changelog see

3.1 pfsspy Package

3.1.1 Functions

carr_cea_wcs_header(dtime, shape)	Create a Carrington WCS header for a Cylindrical Equal Area (CEA) projection.
load_output(file)	Load a saved output file.
pfss(input)	Compute PFSS model.

carr_cea_wcs_header

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

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

dtime [datetime, None] Datetime to associate with the map.

shape [tuple] Map shape. The first entry should be number of points in longitude, the second in latitude.

References

load_output

pfsspy.**load_output**(*file*)

Load a saved output file.

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

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

Returns

Return type *Output*

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

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 $p = \phi$).

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

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

Returns **out**

Return type *Output*

3.1.2 Classes

Grid(ns, nphi, nr, rss)	Grid on which the solution is calculated.
Input(br, nr, rss)	Input to PFSS modelling.
Output(alr, als, alp, grid[, input_map])	Output of PFSS modelling.

Grid

class pfsspy.**Grid**(*ns*, *nphi*, *nr*, *rss*)

Bases: object

Grid on which the solution is calculated.

The grid is evenly spaced in (cos(theta), phi, log(r)). See *pfsspy.coords* for more information.

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).

Input

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

Input to PFSS modelling.

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

Parameters

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

Attributes Summary

map	sunpy.map.GenericMap representation of the input.

Attributes Documentation

map
sunpy.map.GenericMap representation of the input.

Output

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

Output of PFSS modelling.

Parameters

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

Attributes Summary

al	Vector potential times cell edge lenghts.
bc	B on the centres of the cell faces.
bg	B as a (weighted) averaged on grid points.
coordinate_frame	The coordinate frame that the PFSS solution is in.
dtime	
source_surface_br	Br on the source surface.
source_surface_pils	Coordinates of the polarity inversion lines on the source surface.

Methods Summary

save(file)	Save the output to file.
trace(tracer, seeds)	**param tracer** Field line tracer.

Attributes Documentation

al
Vector potential times cell edge lenghts.

Returns ar*Lr, as*Ls, ap*Lp on cell edges.

bc
B on the centres of the cell faces.

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

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

Return type numpy.ndarray

coordinate_frame
The coordinate frame that the PFSS solution is in.

Notes

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

dtime

source_surface_br
Br on the source surface.

Returns

Return type `sunpy.map.GenericMap`

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

Notes

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

Methods Documentation

save(*file*)
Save the output to file.

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

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

trace(*tracer*, *seeds*)

Parameters

- **tracer** (`tracing.Tracer`) – Field line tracer.

- **seeds** (*astropy.coordinates.SkyCoord*) – Starting coordinates.

3.2 pfsspy.fieldline Module

3.2.1 Classes

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

ClosedFieldLines

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

Bases: *pfsspy.fieldline.FieldLines*

A set of closed field lines.

FieldLine

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

Bases: object

A single magnetic field line.

Parameters

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

Attributes Summary

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

coords
Field line `SkyCoord`.

expansion_factor
Magnetic field expansion factor.

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

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

Return type float

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

polarity
Magnetic field line polarity.

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

Return type int

solar_footpoint
Solar surface magnetic field footpoint.

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

Returns **footpoint**

Return type `SkyCoord`

Notes

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

source_surface_footpoint
Solar surface magnetic field footpoint.

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

Returns **footpoint**

Return type `SkyCoord`

Notes

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

FieldLines

class pfsspy.fieldline.**FieldLines**(*field_lines*)
Bases: object

A collection of *FieldLine*.

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

Attributes Summary

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

Attributes Documentation

closed_field_lines
An *ClosedFieldLines* object containing open field lines.

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

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

open_field_lines
An *OpenFieldLines* object containing open field lines.

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

OpenFieldLines

class pfsspy.fieldline.**OpenFieldLines**(*field_lines*)
Bases: *pfsspy.fieldline.FieldLines*

A set of open field lines.

Attributes Summary

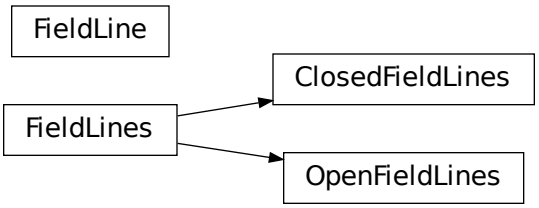
solar_feet	Coordinates of the solar footpoints.
source_surface_feet	Coordinates of the source suface footpoints.

Attributes Documentation

solar_feet
Coordinates of the solar footpoints.

source_surface_feet
Coordinates of the source suface footpoints.

3.2.2 Class Inheritance Diagram



3.3 pfsspy.tracing Module

3.3.1 Classes

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

FortranTracer

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

Tracer using Fortran code.

Parameters

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

Notes

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

Methods Summary

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

Methods Documentation

trace (*seeds*, *output*)

Parameters

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

Returns **streamlines** – Traced field lines.

Return type *FieldLines*

static vector_grid (*output*)

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

PythonTracer

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

Bases: *pfsspy.tracing.Tracer*

Tracer using native python code.

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

Methods Summary

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

Methods Documentation

trace(*seeds*, *output*)

Parameters

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

Returns **streamlines** – Traced field lines.

Return type *FieldLines*

Tracer

class pfsspy.tracing.**Tracer**

Bases: abc.ABC

Abstract base class for a streamline tracer.

Methods Summary

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

Methods Documentation

static cartesian_to_coordinate()

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

static coords_to_xyz(*seeds*, *output*)

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

Parameters

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

abstract trace(*seeds*, *output*)

Parameters

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

Returns **streamlines** – Traced field lines.

Return type *FieldLines*

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

3.4 Changelog

3.4.1 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.

3.4.2 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 `xyz_to_coords`. These new methods convert between `SkyCoord` objects, and Cartesian xyz coordinates of the internal magnetic field grid.

3.4.3 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.

3.4.4 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.

3.4.5 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.

3.4.6 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.

3.4.7 0.3.2

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

3.4.8 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.

3.4.9 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.

3.4.10 0.2.0

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

3.4.11 0.1.5

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

3.4.12 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.

3.4.13 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.

for usage examples see

3.5 pfsspy examples

3.5.1 Using pfsspy

GONG helper functions

```
import os

import numpy as np


def get_gong_map():
    """
    Automatically download and unzip a sample GONG synoptic map.
    """
    if not os.path.exists('190310t0014gong.fits') and not os.path.exists(
↪'190310t0014gong.fits.gz'):
        import urllib.request
        urllib.request.urlretrieve(
            'https://gong2.nso.edu/oQR/zqs/201903/mrzqs190310/mrzqs190310t0014c2215_
↪333.fits.gz',
            '190310t0014gong.fits.gz')

    if not os.path.exists('190310t0014gong.fits'):
        import gzip
        with gzip.open('190310t0014gong.fits.gz', 'rb') as f:
            with open('190310t0014gong.fits', 'wb') as g:
                g.write(f.read())

    return '190310t0014gong.fits'
```

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

Open/closed field map

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

First, import required modules

```
import os

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

import numpy as np
import sunpy.map

import pfsspy
from pfsspy import coords
from pfsspy import tracing
from gong_helpers import get_gong_map
```

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

```
gong_fname = get_gong_map()
```

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

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

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

Set the model parameters

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

Construct the input, and calculate the output solution

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

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

First, set up the tracing seeds

```
r = const.R_sun
# Number of steps in cos(latitude)
nsteps = 30
lon_1d = np.linspace(0, 2 * np.pi, nsteps * 2 + 1)
```

(continues on next page)

(continued from previous page)

```
lat_1d = np.arcsin(np.linspace(-1, 1, nsteps + 1))
lon, lat = np.meshgrid(lon_1d, lat_1d, indexing='ij')
lon, lat = lon*u.rad, lat*u.rad
seeds = SkyCoord(lon.ravel(), lat.ravel(), r, frame=output.coordinate_frame)
```

Trace the field lines

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

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

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

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

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

plt.show()
```

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

Dipole source solution

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

First, import required modules

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

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

```
nphi = 360
ns = 180

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

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

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


br = dipole_Br(1, s)
```

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

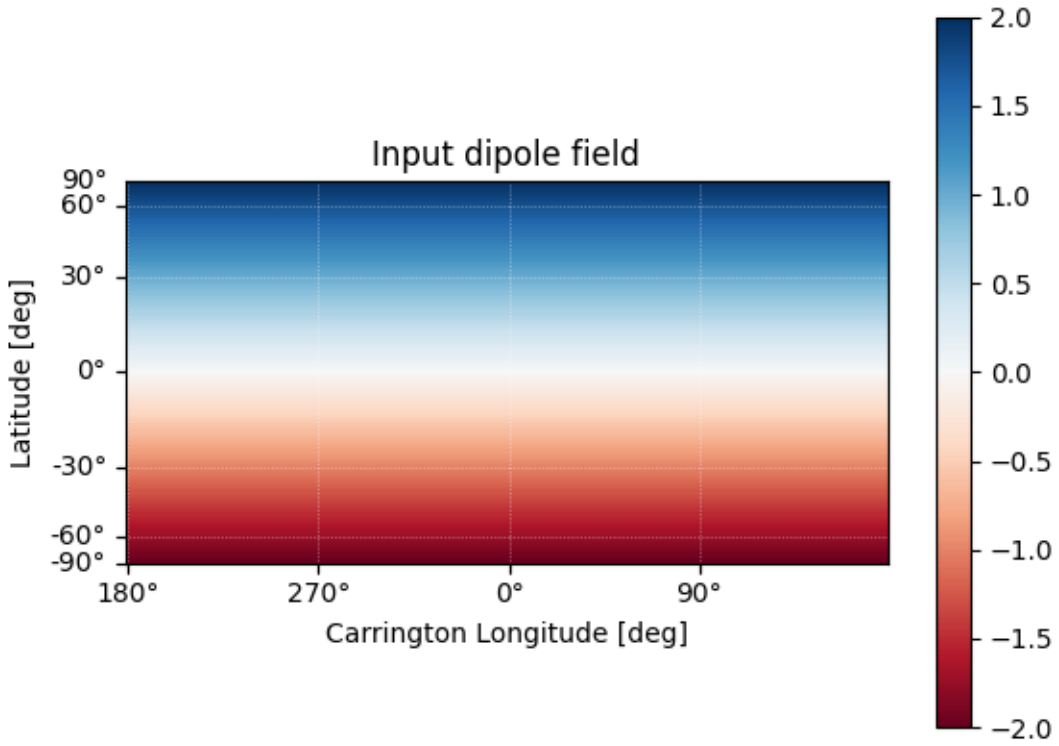
```
nrho = 30
rss = 2.5
```

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

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

Using the Input object, plot the input field

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



Out:

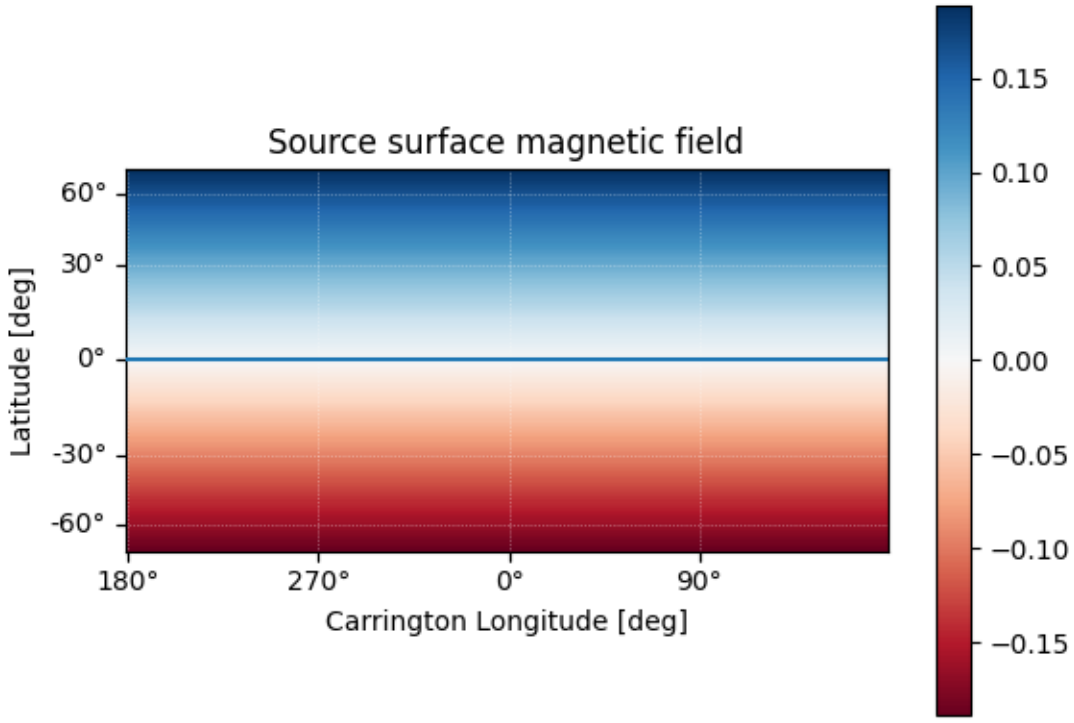
```
Text(0.5, 1.0, 'Input dipole field')
```

Now calculate the PFSS solution.

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

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

```
ss_br = output.source_surface_br

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

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



Out:

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

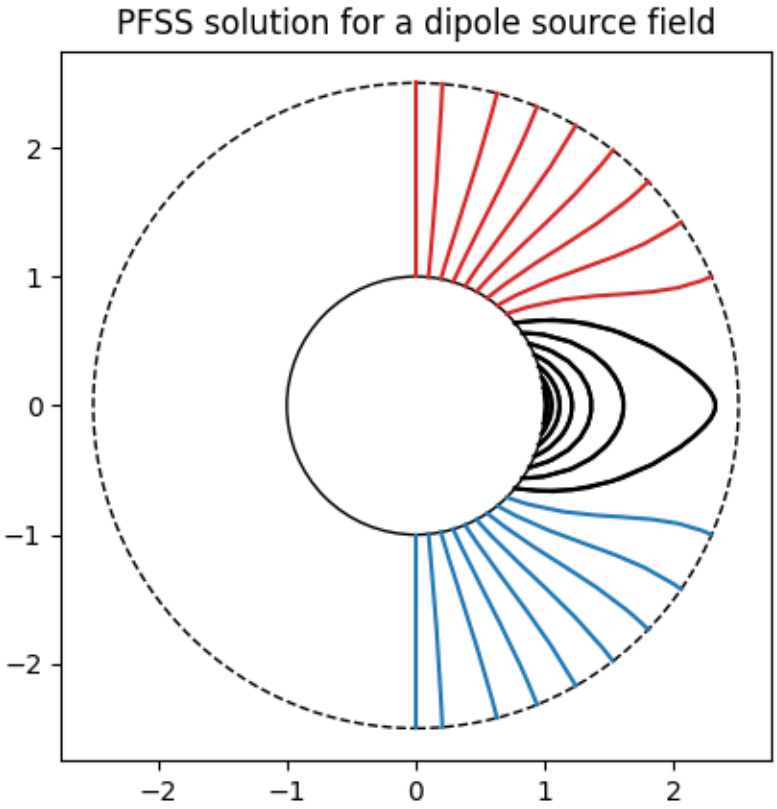
Finally, using the 3D magnetic field solution we can trace some field lines. In this case 32 points equally spaced in theta are chosen and traced from the source surface outwards.

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

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

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

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

(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), input.grid.rss, color='k', linestyle='--',
                           fill=False))
ax.set_title('PFSS solution for a dipole source field')
plt.show()
```



Total running time of the script: (0 minutes 5.474 seconds)

Overplotting field lines on AIA maps

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

First, we import the required modules

```
from datetime import datetime
import os

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

(continues on next page)

(continued from previous page)

```
import sunpy.map
import sunpy.io.fits

import pfsspy
import pfsspy.coords as coords
import pfsspy.tracing as tracing
from gong_helpers import get_gong_map
```

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

```
gong_fname = get_gong_map()
```

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

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

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

Load the corresponding AIA 193 map

```
if not os.path.exists('AIA20190310.fits'):
    import urllib.request
    urllib.request.urlretrieve(
        'http://jsoc2.stanford.edu/data/aia/synoptic/2019/03/10/H0000/AIA20190310_
↪0000_0193.fits',
        'AIA20190310.fits')

aia = sunpy.map.Map('AIA20190310.fits')
dtime = aia.date
```

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

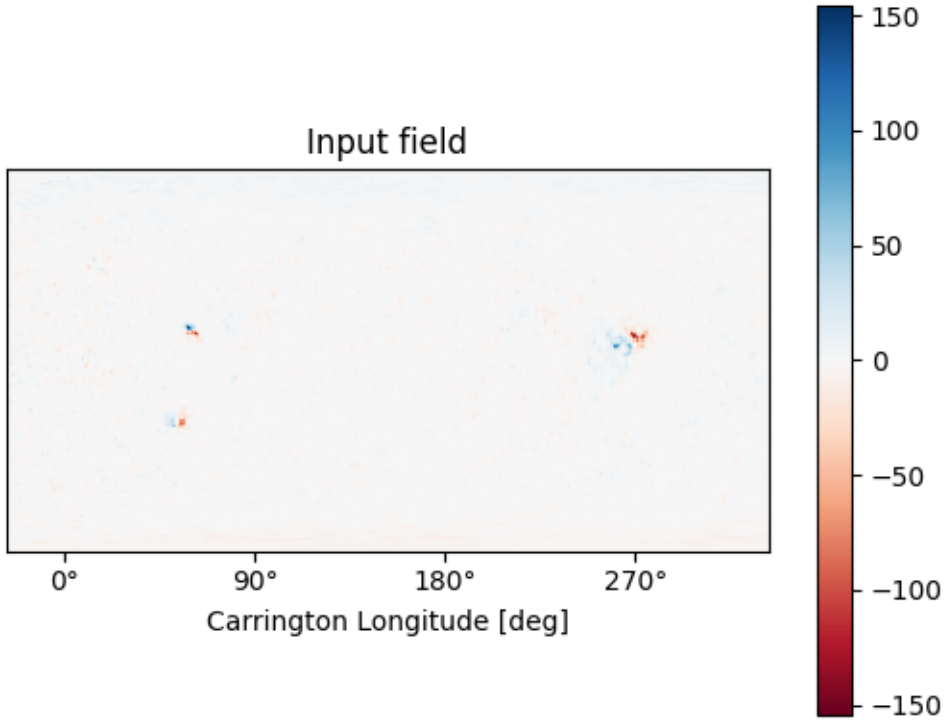
```
nrho = 25
rss = 2.5
```

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

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

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

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

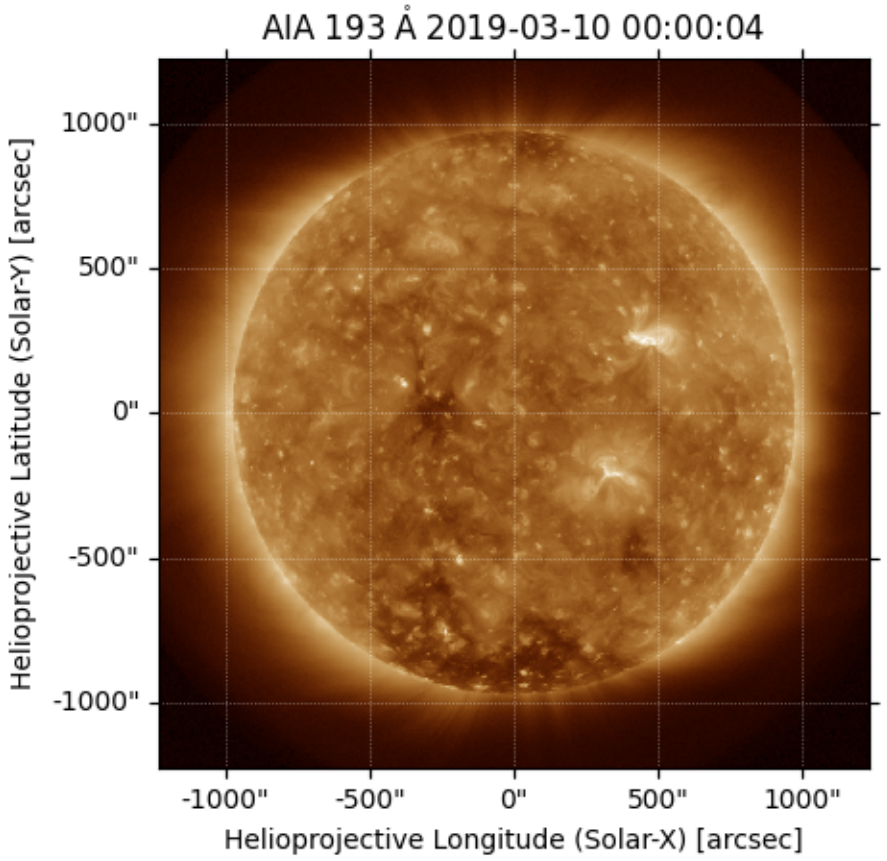


Out:

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

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

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



Out:

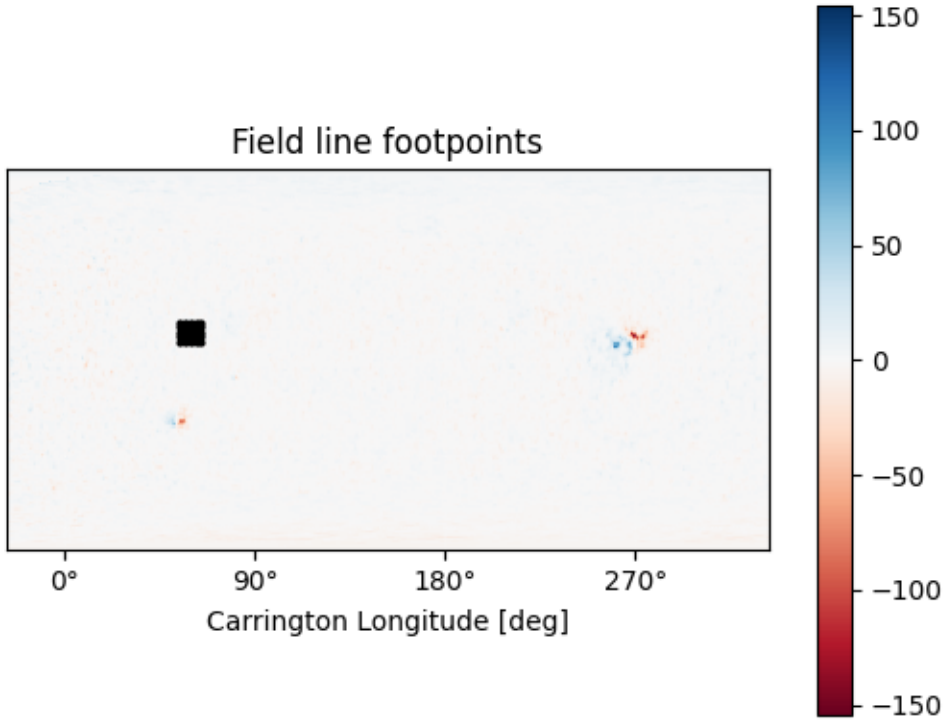
```
<matplotlib.image.AxesImage object at 0x7fbe91089748>
```

Now we construct a 10 x 10 grid of footpoitns to trace some magnetic field lines from.

```
s, phi = np.meshgrid(np.linspace(0.1, 0.2, 5),
                     np.deg2rad(np.linspace(55, 65, 5)))
lat = np.arcsin(s) * u.rad
lon = phi * u.rad
seeds = SkyCoord(lon.ravel(), lat.ravel(), 1.01 * const.R_sun,
                 frame=gong_map.coordinate_frame)
```

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

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

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

ax.set_title('Field line footpoints')
ax.set_ylim(bottom=0)
```



Out:

```
(0.0, 179.5)
```

Compute the PFSS solution from the GONG magnetic field input

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

Trace field lines from the footpoints defined above.

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

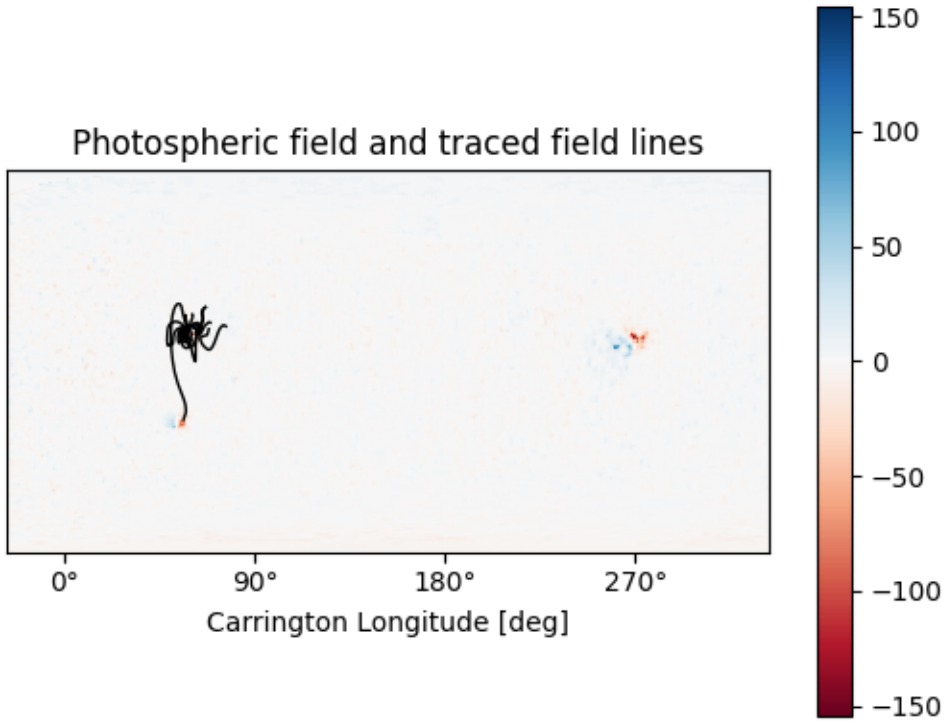
Plot the input GONG magnetic field map, along with the traced mangetic field lines.

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

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

# ax.set_xlim(55, 65)
```

(continues on next page)

(continued from previous page)

```
# ax.set_ylim(0.1, 0.25)
ax.set_title('Photospheric field and traced field lines')
```

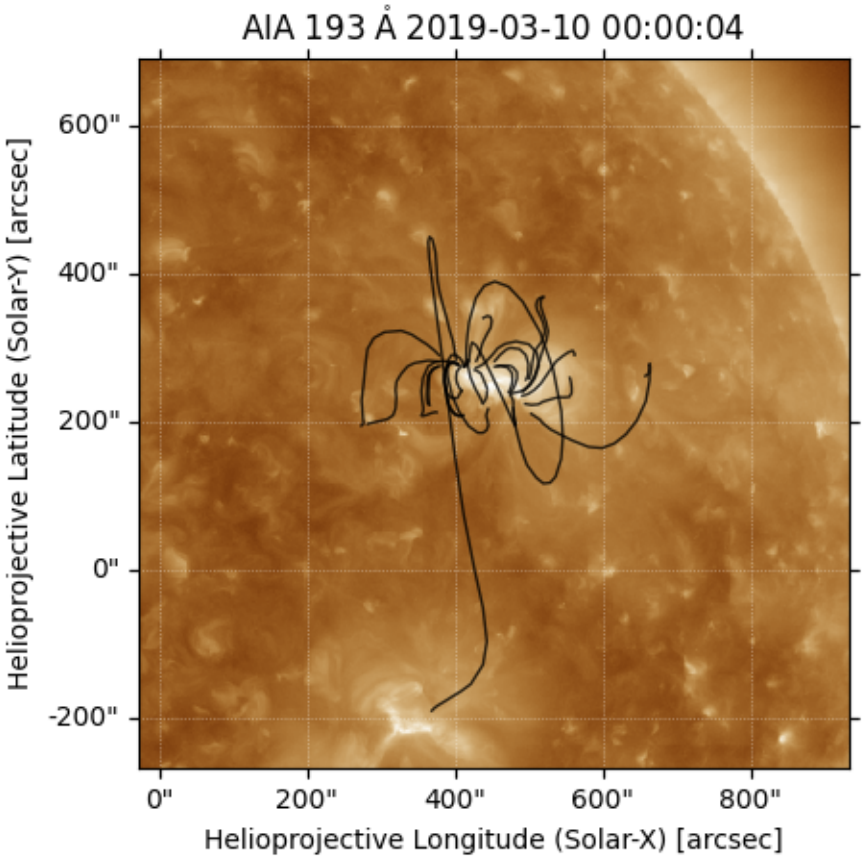


Out:

```
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)
transform = ax.get_transform('world')
aia.plot(ax)
for fline in flines:
    ax.plot_coord(fline.coords, alpha=0.8, linewidth=1, color='black')

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

# sphinx_gallery_thumbnail_number = 5
```



Total running time of the script: (0 minutes 12.455 seconds)

GONG PFSS extrapolation

Calculating PFSS solution for a GONG synoptic magnetic field map.

First, import required modules

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

import pfsspy
from pfsspy import coords
from pfsspy import tracing
from gong_helpers import get_gong_map
```

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

```
gong_fname = get_gong_map()
```

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

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

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

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

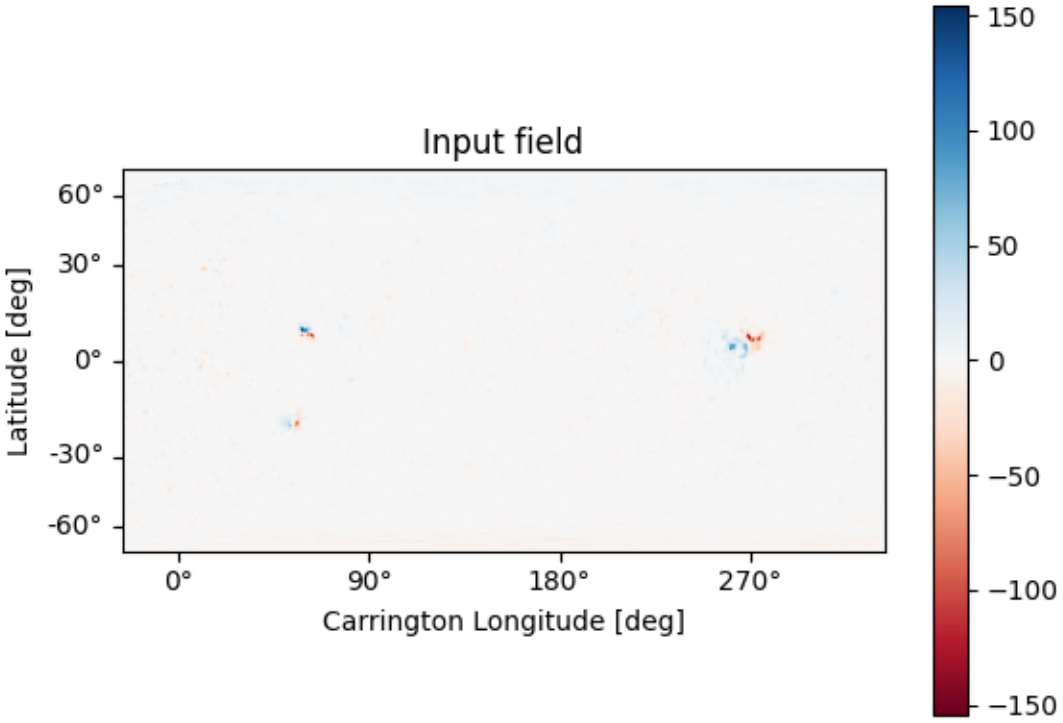
```
nrho = 35
rss = 2.5
```

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

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


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

Using the Input object, plot the input field

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

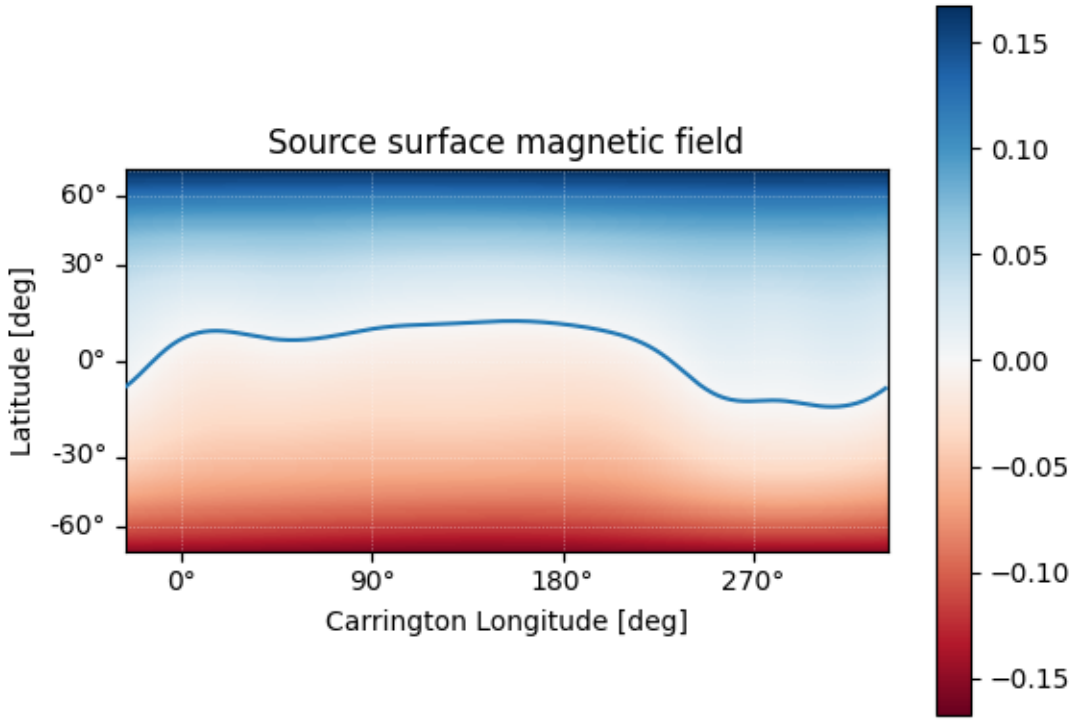


Now calculate the PFSS solution, and plot the polarity inversion line.

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

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

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

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



Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/pfsspy/envs/0.5.2/lib/python3.7/site-
↪packages/astropy/wcs/wcs.py:687: FITSFixedWarning: 'datfix' made the change 'Set␣
↪DATE-REF to '1858-11-17' from MJD-REF'.
  FITSFixedWarning)
/home/docs/checkouts/readthedocs.org/user_builds/pfsspy/envs/0.5.2/lib/python3.7/site-
↪packages/astropy/wcs/wcs.py:687: FITSFixedWarning: 'datfix' made the change 'Set␣
↪DATE-REF to '1858-11-17' from MJD-REF'.
  FITSFixedWarning)
/home/docs/checkouts/readthedocs.org/user_builds/pfsspy/envs/0.5.2/lib/python3.7/site-
↪packages/astropy/wcs/wcs.py:687: FITSFixedWarning: 'datfix' made the change 'Set␣
↪DATE-REF to '1858-11-17' from MJD-REF'.
  FITSFixedWarning)
```

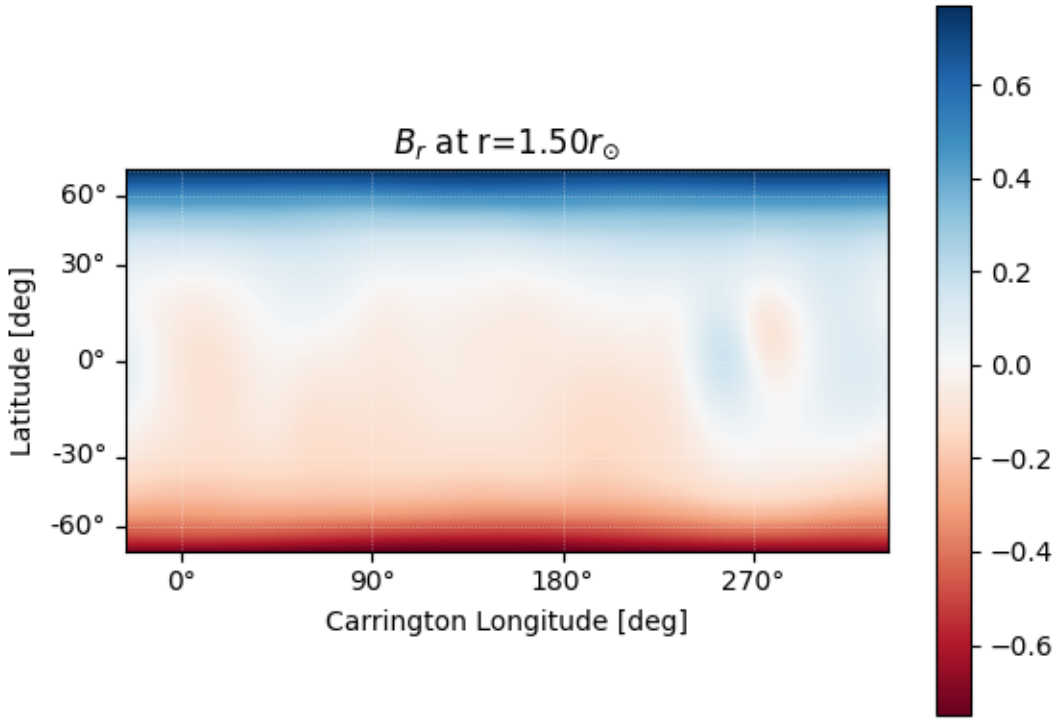
It is also easy to plot the magnetic field at an arbitrary height within the PFSS solution.

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

# Create the figure and axes
```

(continues on next page)

(continued from previous page)

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

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



Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/pfsspy/envs/0.5.2/lib/python3.7/site-
↪packages/astropy/wcs/wcs.py:687: FITSFixedWarning: 'datfix' made the change 'Set␣
↪DATE-REF to '1858-11-17' from MJD-REF'.
  FITSFixedWarning)
```

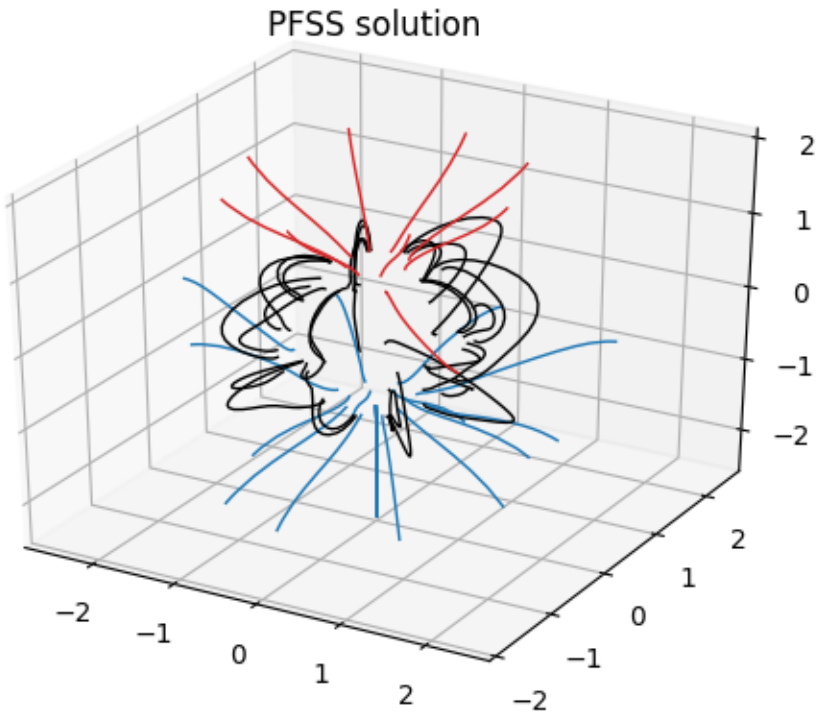
Finally, using the 3D magnetic field solution we can trace some field lines. In this case 64 points equally gridded in theta and phi are chosen and traced from the source surface outwards.

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

tracer = tracing.PythonTracer()
```

(continues on next page)

(continued from previous page)

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

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

field_lines = tracer.trace(seeds, output)

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

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

# sphinx_gallery_thumbnail_number = 4
```



Total running time of the script: (0 minutes 8.623 seconds)

3.5.2 Finding data

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

HMI data

How to search for HMI data.

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

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

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

First import the required modules

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

Set up the search.

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

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

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

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

Out:

```
Results from 1 Provider:

134 Results from the JSOCClient:
     T_REC      TELESCOP  INSTRUME WAVELNTH CAR_ROT
     str15        str7      str9   float64   int64
--------------- -------- --------- -------- -------
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2097
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2098
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2099
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2100
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2101
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2102
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2103
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2104
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2105
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2106
            ...      ...       ...      ...     ...
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2221
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2222
```

(continues on next page)

(continued from previous page)

```
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2223
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2224
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2225
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2226
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2227
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2228
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2229
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2230
```

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

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

Out:

```
Results from 1 Provider:

1 Results from the JSOCClient:
     T_REC      TELESCOP  INSTRUME WAVELNTH CAR_ROT
     str15        str7      str9   float64   int64
--------------- -------- --------- -------- -------
Invalid KeyLink  SDO/HMI HMI_SIDE1   6173.0    2210
```

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

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

Out:

```
Export request pending. [id="JSOC_20200527_2024_X_IN", status=2]
Waiting for 0 seconds...
2 URLs found for download. Full request totalling 2MB

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


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

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


hmi.mrsynop_small_720s.2210.epts.fits:   0%|          | 100/1.05M [00:00<24:17, 717B/
↪s][A[A

hmi.mrsynop_small_720s.2210.synopMr.fits:   0%|          | 100/1.05M [00:00<24:13,␣
↪719B/s][A


hmi.mrsynop_small_720s.2210.epts.fits:   2%|1         | 19.4k/1.05M [00:00<16:43, 1.
↪02kB/s][A[A
```

(continues on next page)

(continued from previous page)

```
hmi.mrsynop_small_720s.2210.synopMr.fits:   2%|1          | 19.4k/1.05M [00:00<16:40,␣
↪1.03kB/s][A


hmi.mrsynop_small_720s.2210.epts.fits:   5%|4          | 50.9k/1.05M [00:00<11:22, 1.
↪46kB/s][A[A

hmi.mrsynop_small_720s.2210.synopMr.fits:   5%|4          | 50.9k/1.05M [00:00<11:20,␣
↪1.46kB/s][A


hmi.mrsynop_small_720s.2210.epts.fits:   9%|9          | 96.6k/1.05M [00:00<07:36, 2.
↪08kB/s][A[A

hmi.mrsynop_small_720s.2210.synopMr.fits:   9%|9          | 95.1k/1.05M [00:00<07:36,␣
↪2.08kB/s][A


hmi.mrsynop_small_720s.2210.epts.fits:  15%|#5         | 159k/1.05M [00:00<04:58, 2.
↪96kB/s] [A[A

hmi.mrsynop_small_720s.2210.synopMr.fits:  15%|#4         | 157k/1.05M [00:00<04:59, 2.
↪97kB/s] [A


hmi.mrsynop_small_720s.2210.epts.fits:  24%|##3        | 246k/1.05M [00:00<03:09, 4.
↪23kB/s][A[A

hmi.mrsynop_small_720s.2210.synopMr.fits:  23%|##2        | 239k/1.05M [00:00<03:10, 4.
↪24kB/s][A


hmi.mrsynop_small_720s.2210.epts.fits:  34%|###4       | 358k/1.05M [00:00<01:54, 6.
↪02kB/s][A[A

hmi.mrsynop_small_720s.2210.synopMr.fits:  34%|###3       | 352k/1.05M [00:00<01:54, 6.
↪04kB/s][A


hmi.mrsynop_small_720s.2210.epts.fits:  49%|####8      | 509k/1.05M [00:01<01:02, 8.
↪59kB/s][A[A

hmi.mrsynop_small_720s.2210.synopMr.fits:  49%|####8      | 509k/1.05M [00:01<01:02, 8.
↪61kB/s][A


hmi.mrsynop_small_720s.2210.epts.fits:  66%|######5    | 689k/1.05M [00:01<00:29, 12.
↪2kB/s][A[A

hmi.mrsynop_small_720s.2210.synopMr.fits:  67%|######6    | 696k/1.05M [00:01<00:28,␣
↪12.3kB/s][A


hmi.mrsynop_small_720s.2210.epts.fits:  90%|######### | 942k/1.05M [00:01<00:05, 17.
↪4kB/s][A[A
```

(continues on next page)

(continued from previous page)

```
hmi.mrsynop_small_720s.2210.synopMr.fits:  87%|########7 | 913k/1.05M [00:01<00:07,␣
↪17.5kB/s][A



↪     [A[A                                                                    ␣
Files Downloaded:  50%|#####     | 1/2 [00:01<00:01,  1.56s/file]


↪       [A                                                                    ␣
Files Downloaded: 100%|##########| 2/2 [00:01<00:00,  1.27file/s]
['/home/docs/sunpy/data/hmi.mrsynop_small_720s.2210.epts.fits', '/home/docs/sunpy/
↪data/hmi.mrsynop_small_720s.2210.synopMr.fits']
```

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

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

Out:

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

Total running time of the script: (0 minutes 16.321 seconds)

ADAPT helper functions

```
import os


def example_adapt_map():
    import urllib.request
    urllib.request.urlretrieve(
        'https://gong.nso.edu/adapt/maps/gong/2020/adapt40311_03k012_202001010000_
↪i00005600n1.fts.gz',
        'adapt20200101.fts.gz'
    )

    if not os.path.exists('adapt20200101.fts'):
        import gzip
        with gzip.open('adapt20200101.fts.gz', 'rb') as f:
            with open('adapt20200101.fts', 'wb') as g:
                g.write(f.read())

    return 'adapt20200101.fts'
```

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

Parsing ADAPT Ensemble .fits files

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

Necessary imports

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

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

```
adapt_fname = example_adapt_map()
```

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

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

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

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

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

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

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

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

`adapt_map_sequence` is now a list of our individual adapt realizations. Note the `peek()` and `plot()` methods of `MapSequence` returns instances of `sunpy.visualization.MapSequenceAnimator` and `matplotlib.animation.FuncAnimation1` instances. 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 ii,aMap in enumerate(adaptMapSequence) :
    ax=fig.add_subplot(gs[ii],projection=aMap)
    aMap.plot(axes=ax,cmap='bwr',vmin=-2,vmax=2,title=f"Realization {1+ii:02d}")
plt.tight_layout(pad=5,h_pad=2)
plt.show()
```

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

3.5.3 pfsspy information

Examples showing how the internals of pfsspy work.

pfsspy magnetic field grid

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

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

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

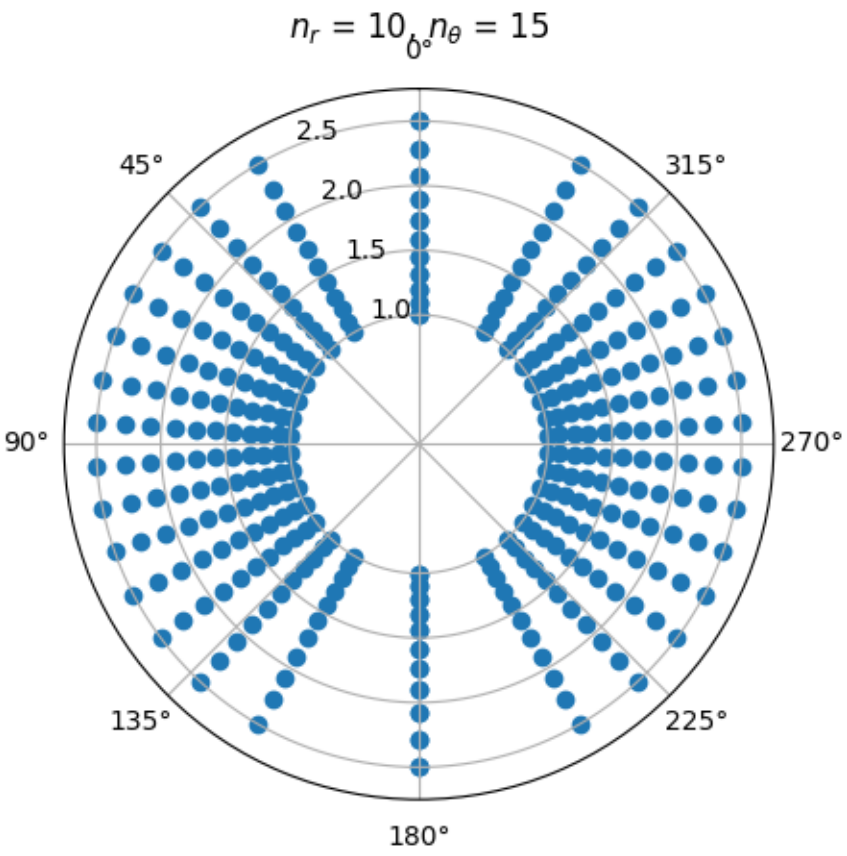
```
r_edges = np.exp(grid.rg)
theta_edges = np.arccos(grid.sg)
```

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 0.181 seconds)

Tracer performance

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

```
import timeit

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

import pfsspy
```

Create a dipole map

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

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

(continues on next page)

(continued from previous page)

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


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


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

Trace some field lines

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

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

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

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

Plot the results

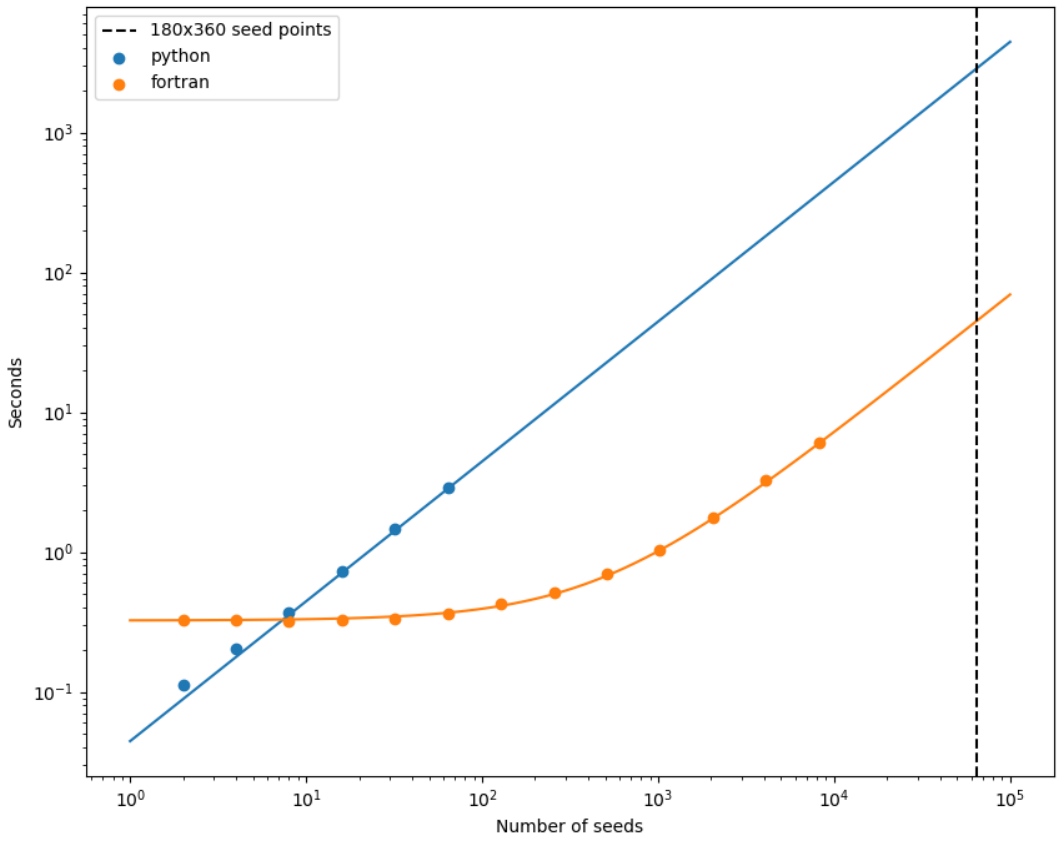
```
fig, ax = plt.subplots()
ax.scatter(nseeds[1:len(times[0])], times[0][1:], label='python')
ax.scatter(nseeds[1:], times[1][1:], label='fortran')

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

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



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

for the helper modules (behind the scense!) see

3.6 Helper modules

3.6.1 pfsspy.coords Module

Helper functions for coordinate transformations used in the PFSS domain.

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

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

The transformation between cartesian coordinates used by the tracer and the above coordinates is given by

- $x = r \sin(\theta) \cos(\phi)$
- $y = r \sin(\theta) \sin(\phi)$
- $z = r \cos(\theta)$

Functions

cart2sph(x, y, z)	Convert cartesian coordinates to spherical coordinates.
cart2strum(x, y, z)	Convert cartesian coordinates to strumfric coordinates.
sph2cart(r, theta, phi)	Convert spherical coordinates to cartesian coordinates.
strum2cart(rho, s, phi)	Convert strumfric coordinates to cartesian coordinates.

cart2sph

pfsspy.coords.**cart2sph**(*x*, *y*, *z*)

Convert cartesian coordinates to spherical coordinates.

Returns

- *r*
- *theta*
- *phi*

cart2strum

pfsspy.coords.**cart2strum**(*x*, *y*, *z*)

Convert cartesian coordinates to strumfric coordinates.

Returns

- *rho*
- *s*
- *phi*

sph2cart

pfsspy.coords.**sph2cart**(*r*, *theta*, *phi*)
Convert spherical coordinates to cartesian coordinates.

strum2cart

pfsspy.coords.**strum2cart**(*rho*, *s*, *phi*)
Convert strumfric coordinates to cartesian coordinates.

and for a quick reference guide to synoptic map FITS conventions see

3.7 Synoptic map FITS conventions

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

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

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

3.7.1 Cylindrical equal area projection

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

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

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

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

S

T

V