pfsspy Documentation

pfsspy contributors

Jan 02, 2020

CONTENTS

1 Improving performance 3

2 Citing 5

3 Code reference 7

4 Indices and tables 41

Python Module Index 43

Index 45

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

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

load_output

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

Load a saved output file.

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

Parameters

file [str, file, Path] File to load.

Returns

Output

pfss

pfsspy.**pfss**(*input*)

Compute PFSS model.

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

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

Parameters

input [*Input*] Input parameters.

Returns

out [*Output*]

3.1.2 Classes

Grid(ns, nphi, nr, rss)	Grid on which the solution is calculated.
Input(br, nr, rss[, dtime])	Input to PFSS modelling.
Output(alr, als, alp, grid[, dtime])	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*, *dtime=None*)
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 [2D array, sunpy.map.Map] Boundary condition of radial magnetic field at the inner surface. If a SunPy map is automatically extracted as map.data with *no* processing.

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.

dtime [datetime, optional] Datetime at which the input map was measured. If given it is attached to the output and any field lines traced from the output.

Methods Summary

plot_input(self[, ax])	Plot a 2D image of the magnetic field boundary condition.

Methods Documentation

plot_input(*self*, *ax=None*, ***kwargs*)
Plot a 2D image of the magnetic field boundary condition.

Parameters

ax [Axes] Axes to plot to. If None, creates a new figure.

Output

class pfsspy.**Output**(*alr*, *als*, *alp*, *grid*, *dtime=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.

dtime [datetime, optional] Datetime at which the input was measured.

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.
source_surface_br	Br on the source surface.

Methods Summary

plot_pil(self[, ax])	Plot the polarity inversion line on the source surface.
plot_source_surface(self[, ax])	Plot a 2D image of the magnetic field at the source surface.
save(self, file)	Save the output to file.
trace(self, tracer, seeds)	**Parameters**

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

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

source_surface_br
Br on the source surface.

Methods Documentation

plot_pil(*self*, *ax=None*, ***kwargs*)
Plot the polarity inversion line on the source surface.

The PIL is where Br = 0.

Parameters

ax [Axes] Axes to plot to. If `None`, creates a new figure.

****kwargs :** Keyword arguments are handed to *ax.contour*.

plot_source_surface(*self*, *ax=None*, ***kwargs*)
Plot a 2D image of the magnetic field at the source surface.

Parameters

ax [Axes] Axes to plot to. If `None`, creates a new figure.

kwargs : Additional keyword arguments are handed to *pcolormesh* that renders the source surface. A useful option here is handing `rasterized=True` to rasterize the image.

save(*self*, *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(*self*, *tracer*, *seeds*)

Parameters

tracer [tracing.Tracer]

seeds [(n, 3) shaped array] Starting coordinates, in cartesian coordinates. *pfsspy.coords* can be used to convert from spherical coordinates to cartesian coordinates and vice versa.

3.2 pfsspy.fieldline Module

3.2.1 Classes

ClosedFieldLines(field_lines)	A set of closed field lines.
FieldLine(x, y, z, dtime, 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*, *dtime*, *output*)
Bases: object

A single magnetic field line.

Parameters

x, y, z [array] Field line coordinates in a Carrington frame of reference. Must be in units of solar radii.

dtime [astropy.time.Time] Time at which the field line was traced. Needed for transforming the field line coordinates to other coordinate frames.

output [Output] The PFSS output through which this field line was traced.

Attributes

coords [astropy.coordinates.SkyCoord] Field line coordinates.

Attributes Summary

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

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 [float] Field line expansion factor. If field line is closed, returns np.nan.

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 [int] 0 if the field line is closed, otherwise sign(Br) of the magnetic field on the solar surface.

solar_footpoint
Solar surface magnetic field footpoint.

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

Returns

footpoint [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 [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.
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.

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

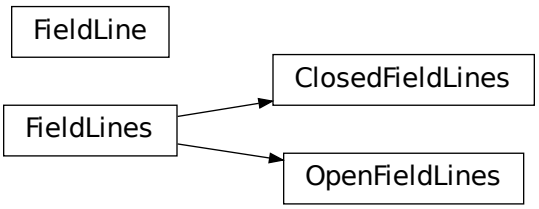
solar_feet	Coordinates of the solar footpoints.
source_surface_feet	Coordinates of the source suface footpoints.

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=0.1*)
Bases: *pfsspy.tracing.Tracer*

Tracer using Fortran code.

Parameters

max_steps: int, optional Maximum number of steps each streamline can take before stopping.

step_size [float, optional] 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(self, seeds, output)	

Parameters

Methods Documentation

trace(*self*, *seeds*, *output*)

Parameters

seeds [(n, 3) array] Coordinaes of the magnetic field seed points, in cartesian coordinates.

output [pfsspy.Output] pfss output.

Returns

streamlines [FieldLines] Traced field lines.

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(self, seeds, output)	**Parameters**

Methods Documentation

trace (*self*, *seeds*, *output*)

Parameters

seeds [(n, 3) array] Coordinaes of the magnetic field seed points, in cartesian coordinates.

output [pfsspy.Output] pfss output.

Returns

streamlines [FieldLines] Traced field lines.

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.
trace(self, seeds, output)	**Parameters**
validate_seeds_shape(seeds)	Check that *seeds* has the right shape.

Methods Documentation

static cartesian_to_coordinate ()

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

abstract trace (*self*, *seeds*, *output*)

Parameters

seeds [(n, 3) array] Coordinaes of the magnetic field seed points, in cartesian coordinates.

output [pfsspy.Output] pfss output.

Returns

streamlines [FieldLines] Traced field lines.

static validate_seeds_shape (*seeds*)

Check that *seeds* has the right shape.

3.4 Changelog

3.4.1 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.

3.4.2 0.3.2

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

3.4.3 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.

3.4.4 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 `footpoints` attribute that returns the footpoint(s) of the field line.

3.4.5 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 Carrignton 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()*

3.4.6 0.1.5

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

3.4.7 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.

3.4.8 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

Note: Click *here* to download the full example code

3.5.1 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')

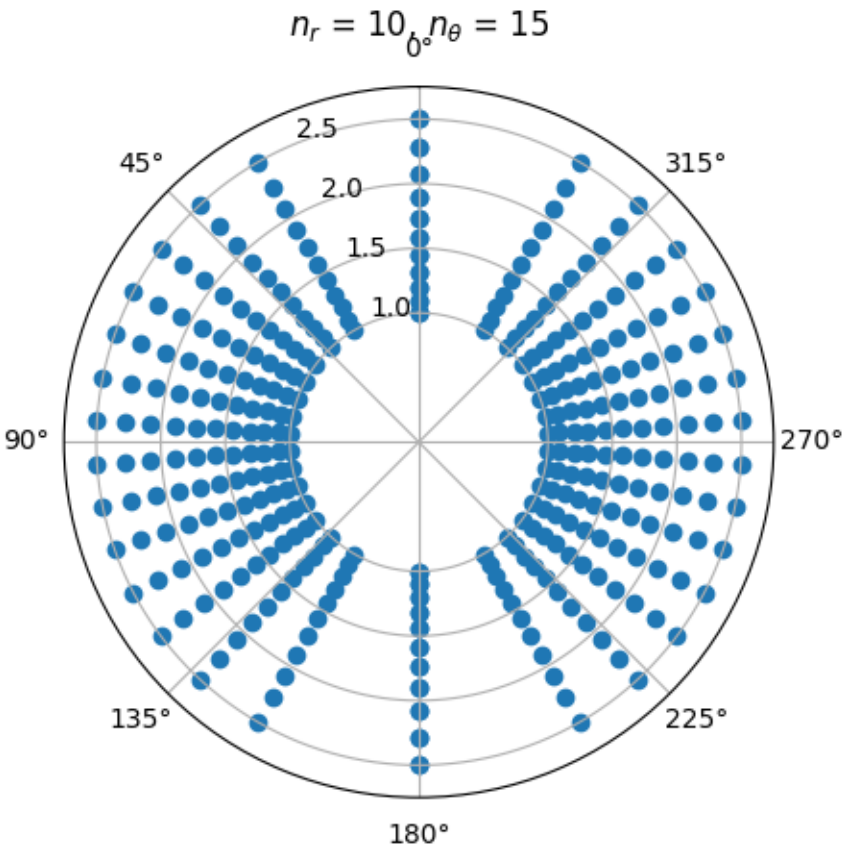
ax.scatter(theta_grid_points, r_grid_points)
ax.scatter(theta_grid_points + np.pi, r_grid_points, color='C0')
```

(continues on next page)

(continued from previous page)

```
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 0.492 seconds)

Note: Click *here* to download the full example code

3.5.2 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 matplotlib.pyplot as plt
import matplotlib.patches as mpatch
import numpy as np
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).T
```

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.

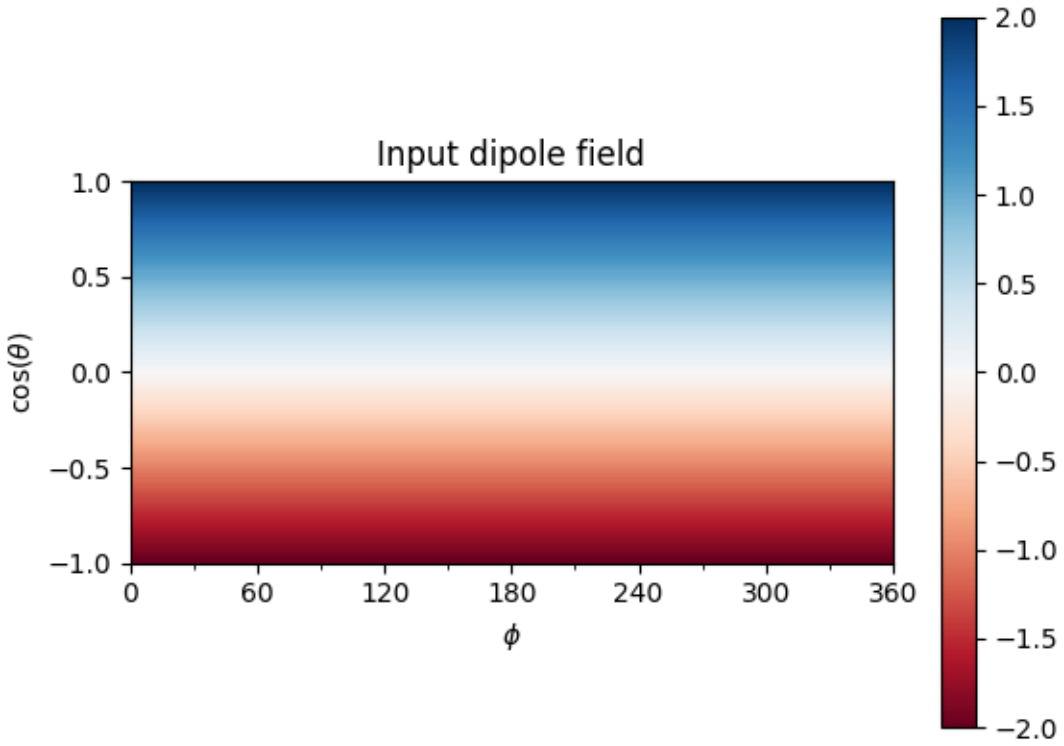
```
nrho = 30
rss = 2.5
```

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

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

Using the Input object, plot the input field

```
fig, ax = plt.subplots()
mesh = input.plot_input(ax)
fig.colorbar(mesh)
ax.set_title('Input dipole field')
```



Out:

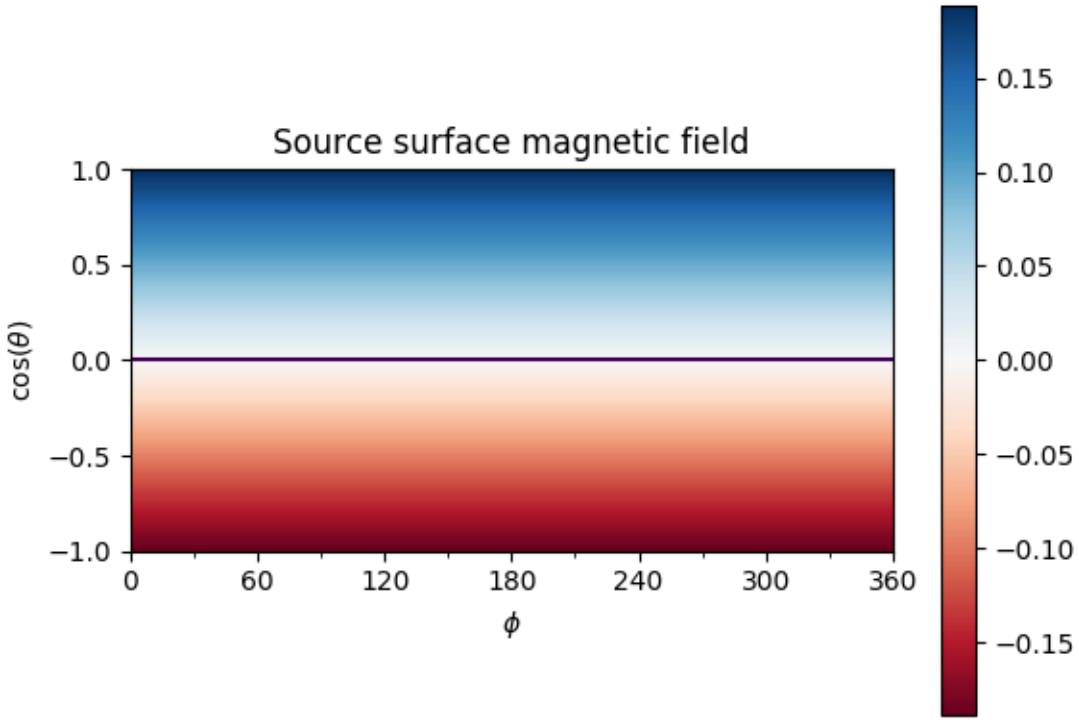
```
Text(0.5, 1.0, 'Input dipole field')
```

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.

```
fig, ax = plt.subplots()
mesh = output.plot_source_surface(ax)
fig.colorbar(mesh)
output.plot_pil(ax)
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
phi = np.pi / 2
r = 1.01
phi = np.pi / 2
theta = np.linspace(0, np.pi, 33)
x0 = np.array(coords.sph2cart(r, theta, phi)).T

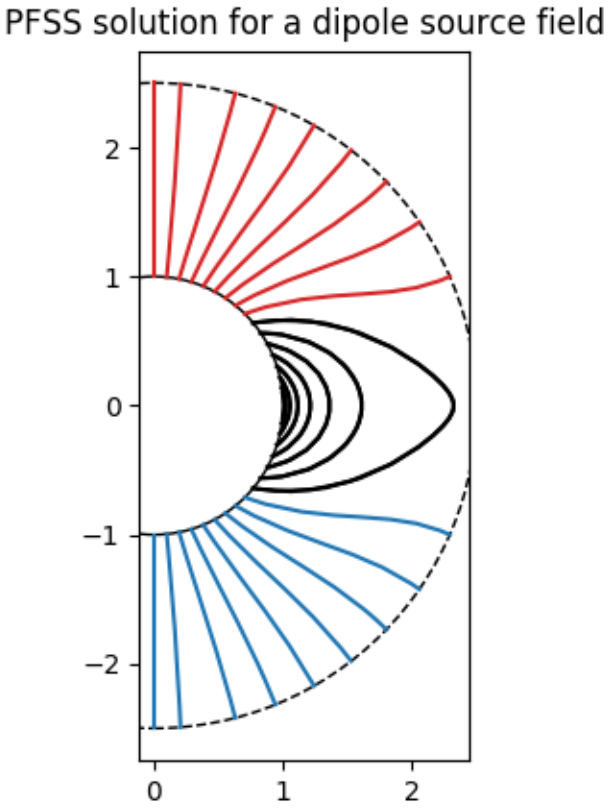
tracer = pfsspy.tracing.PythonTracer()
field_lines = tracer.trace(x0, output)

for field_line in field_lines:
    color = {0: 'black', -1: 'tab:blue', 1: 'tab:red'}.get(field_line.polarity)
    ax.plot(field_line.coords.y / const.R_sun,
            field_line.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 6.617 seconds)

Note: Click *here* to download the full example code

3.5.3 Tracer performance

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

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

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


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


br = dipole_Br(1, theta).T
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)
    for i, tracer in enumerate(tracers):
        if nseed > 64 and i == 0:
            continue
        # tracer.trace(seeds, pfss_output)
        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')

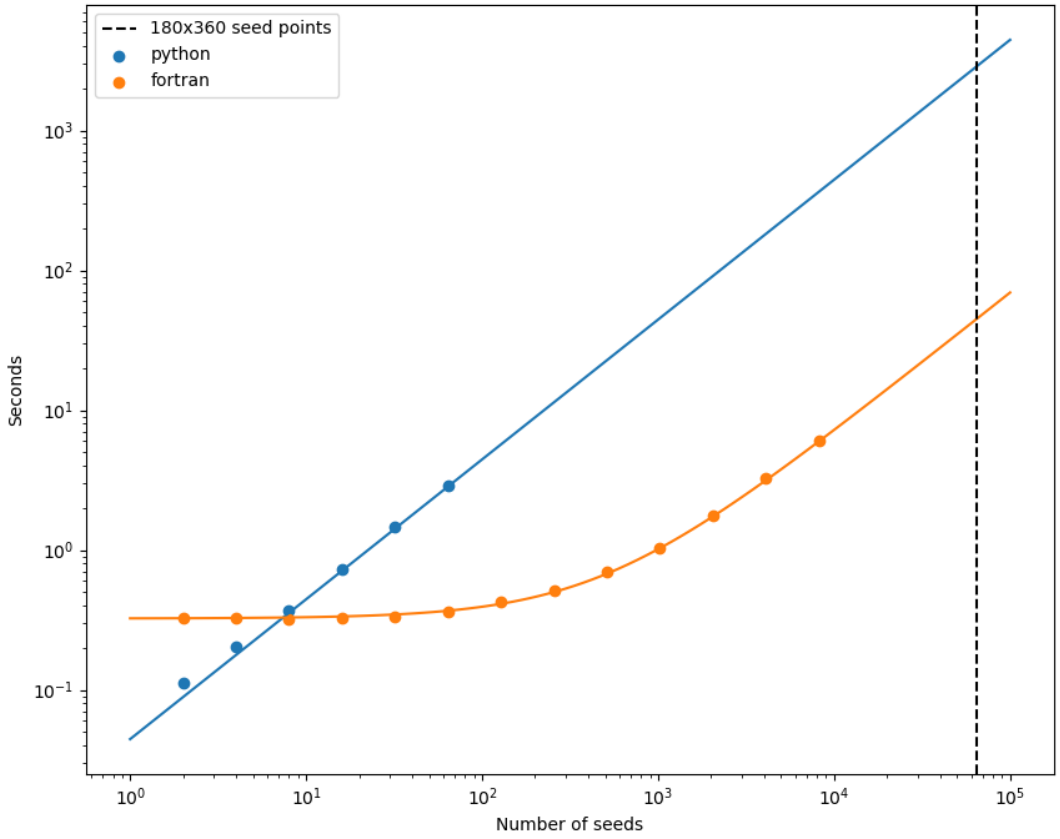
ax.axvline(180 * 360, color='k', linestyle='--', label='180x360 seed points')
```

(continues on next page)

(continued from previous page)

```
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)

Note: Click *here* to download the full example code

3.5.4 Open/closed field map

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

First, import required modules

```
import os
import astropy.constants as const
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
```

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

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!

```
[[br, header]] = sunpy.io.fits.read('190310t0014gong.fits')
br = br - np.mean(br)
```

GONG maps have their LH edge at -180deg in Carrington Longitude, so roll to get it at 0deg. This way the input magnetic field is in a Carrington frame of reference, which matters later when lining the field lines up with the AIA image.

```
br = np.roll(br, header['CRVAL1'] + 180, axis=1)
```

Set the model parameters

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

Construct the input, and calculate the output solution

```
input = pfsspy.Input(br, 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 = 1
# Number of steps in cos(latitude)
nsteps = 90
phi_1d = np.linspace(0, 2 * np.pi, nsteps * 2 + 1)
theta_1d = np.arccos(np.linspace(-1, 1, nsteps + 1))
phi, theta = np.meshgrid(phi_1d, theta_1d, indexing='ij')
seeds = np.array(coords.sph2cart(r, theta.ravel(), phi.ravel())).T
```

Trace the field lines

```
print('Tracing field lines...')
tracer = tracing.FortranTracer()
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, axs = plt.subplots(nrows=2, sharex=True, sharey=True)
ax = axs[0]
input.plot_input(ax)
ax.set_title('Input GONG magnetogram')

ax = axs[1]
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(phi_1d), np.cos(theta_1d), pols, norm=norm, cmap=cmap)

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

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

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

Note: Click *here* to download the full example code

3.5.5 GONG PFSS extrapolation

Calculating PFSS solution for a GONG synoptic magnetic field map.

First, import required modules

```
import os
import astropy.constants as const
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
```

(continues on next page)

(continued from previous page)

```
import sunpy.map

import pfsspy
from pfsspy import coords
from pfsspy import tracing
```

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

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!

```
[[br, header]] = sunpy.io.fits.read('190310t0014gong.fits')
br = br - np.mean(br)
```

GONG maps have their LH edge at -180deg in Carrington Longitude, so roll to get it at 0deg. This way the input magnetic field is in a Carrington frame of reference, which matters later when lining the field lines up with the AIA image.

```
br = np.roll(br, header['CRVAL1'] + 180, axis=1)
```

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.

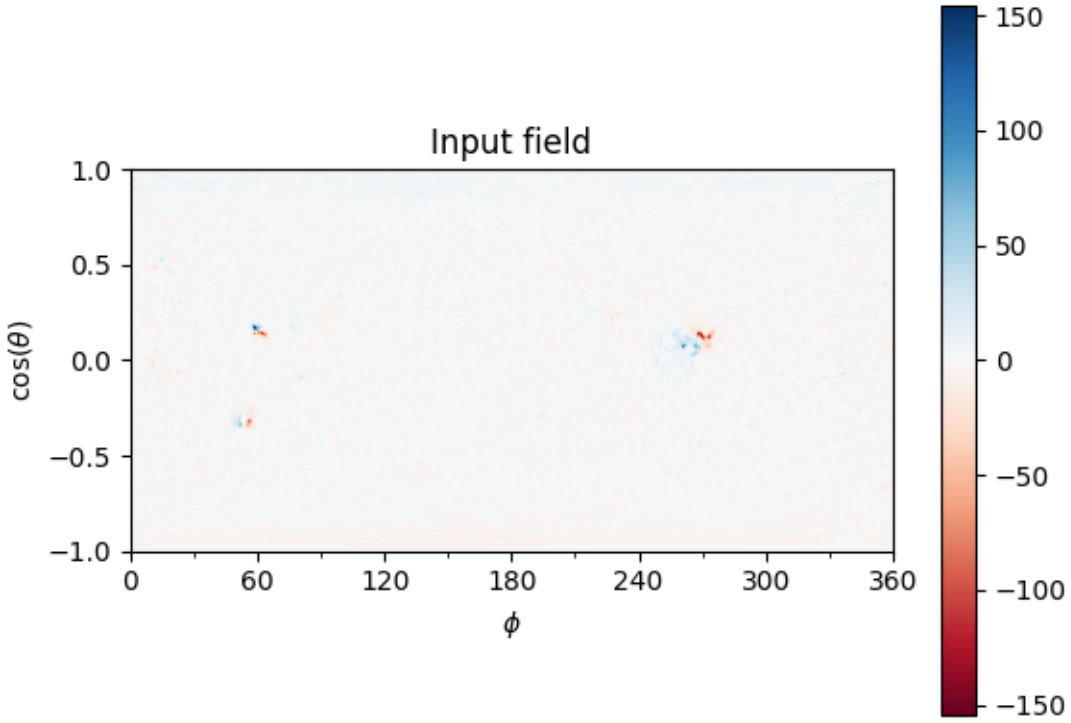
```
nrho = 35
rss = 2.5
```

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(br, nrho, rss)
```

Using the Input object, plot the input field

```
fig, ax = plt.subplots()
mesh = input.plot_input(ax)
fig.colorbar(mesh)
ax.set_title('Input field')
```



Out:

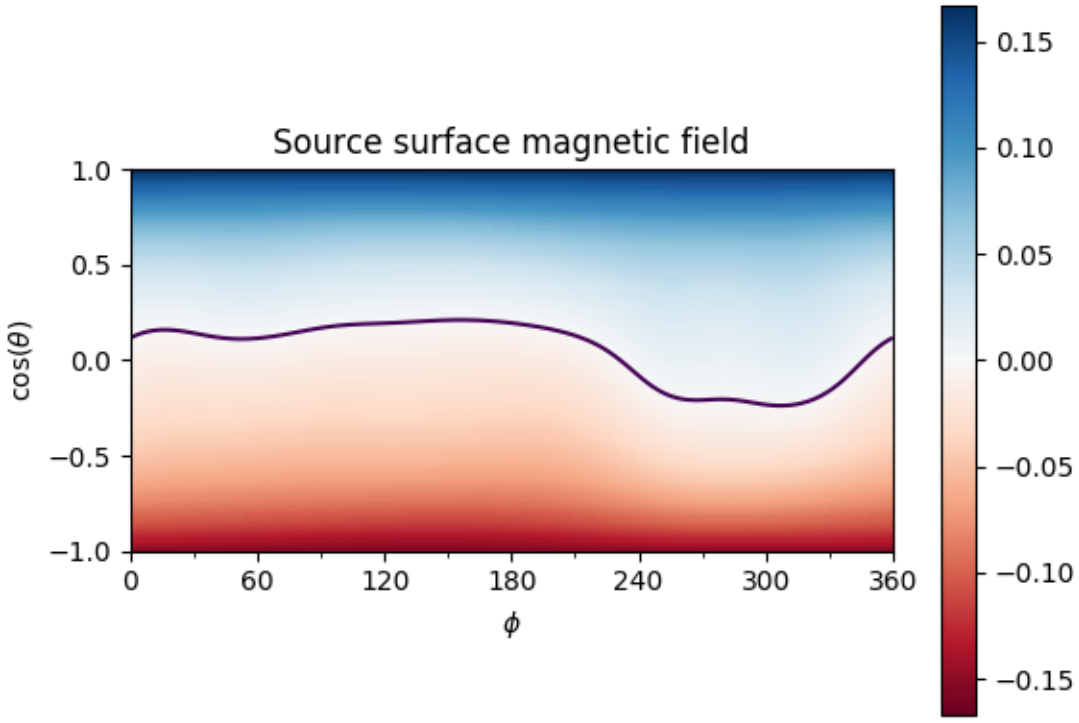
```
Text(0.5, 1.0, 'Input field')
```

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.

```
fig, ax = plt.subplots()
mesh = output.plot_source_surface(ax)
fig.colorbar(mesh)
output.plot_pil(ax)
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 64 points equally gridded in theta and phi are chosen and traced from the source surface outwards.

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

tracer = tracing.PythonTracer()
r = 1.2
theta = np.linspace(0, np.pi, 8, endpoint=False)
phi = np.linspace(0, 2 * np.pi, 8, endpoint=False)
theta, phi = np.meshgrid(theta, phi)
theta, phi = theta.ravel(), phi.ravel()

seeds = np.array(coords.sph2cart(r, theta, phi)).T

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)
    ax.plot(field_line.coords.x / const.R_sun,
            field_line.coords.y / const.R_sun,
            field_line.coords.z / const.R_sun,
```

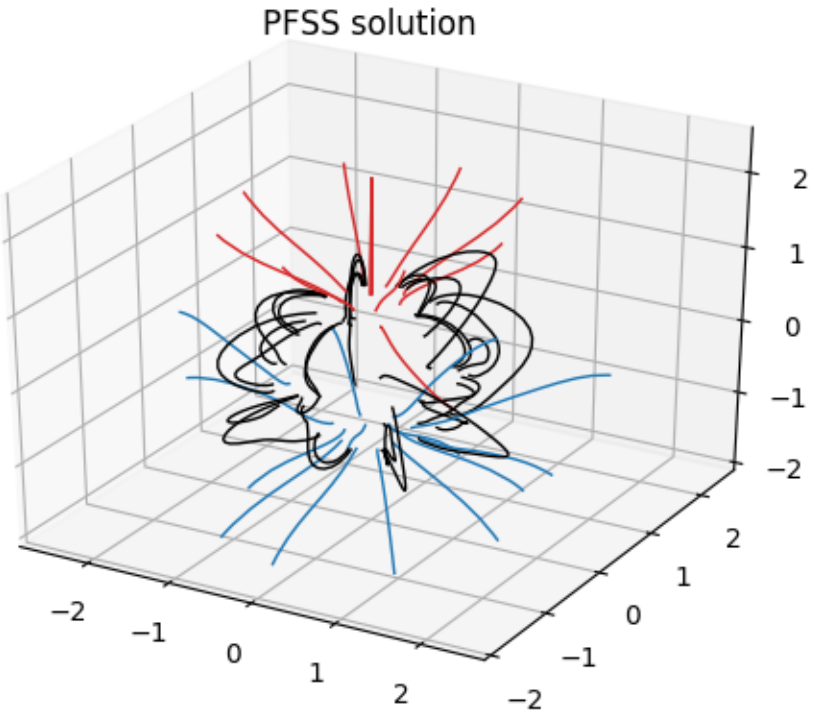
(continues on next page)

(continued from previous page)

```
            color=color, linewidth=1)

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

# sphinx_gallery_thumbnail_number = 3
```



Total running time of the script: (0 minutes 8.378 seconds)

Note: Click *here* to download the full example code

3.5.6 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.units as u
```

(continues on next page)

(continued from previous page)

```
import matplotlib.pyplot as plt
import numpy as np
import sunpy.map
import sunpy.io.fits

import pfsspy
import pfsspy.coords as coords
import pfsspy.tracing as tracing
```

Load a GONG magnetic field map. The map date is 10/03/2019

```
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())
```

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
```

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!

```
[[br, header]] = sunpy.io.fits.read('190310t0014gong.fits')
br = br - np.mean(br)
```

GONG maps have their LH edge at -180deg in Carrington Longitude, so roll to get it at 0deg. This way the input magnetic field is in a Carrington frame of reference, which matters later when lining the field lines up with the AIA image.

```
br = np.roll(br, header['CRVAL1'] + 180, axis=1)
```

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.

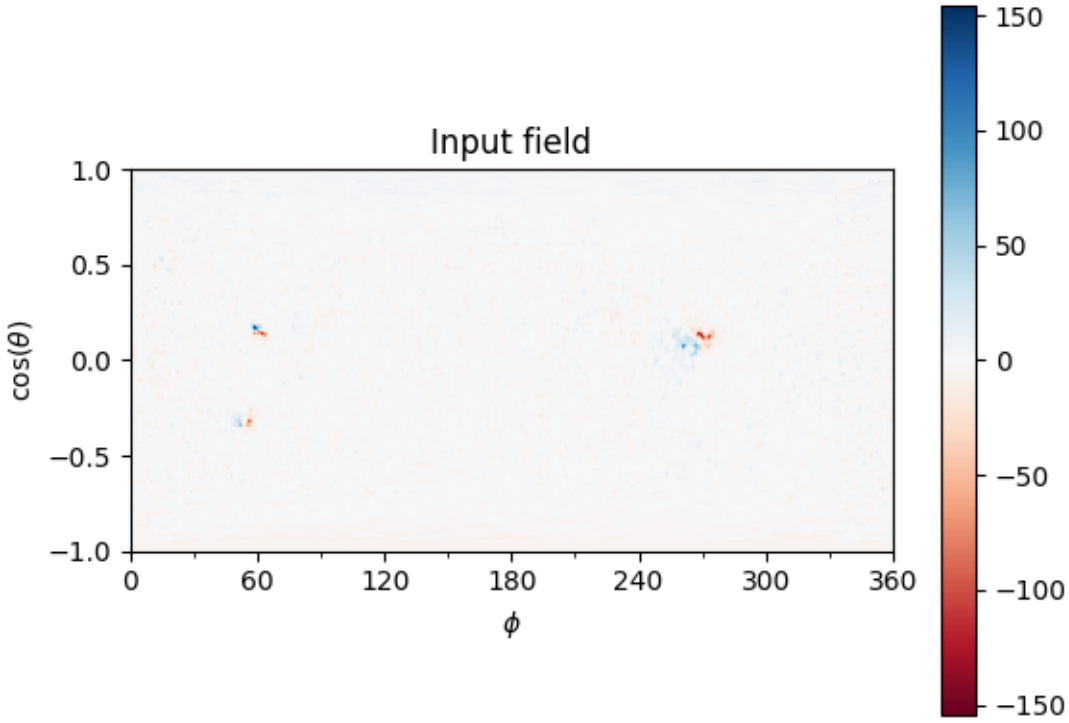
```
nrho = 50
rss = 2.5
```

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(br, nrho, rss, dtime=dtime)
```

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

```
fig, ax = plt.subplots()
mesh = input.plot_input(ax)
fig.colorbar(mesh)
ax.set_title('Input field')
```

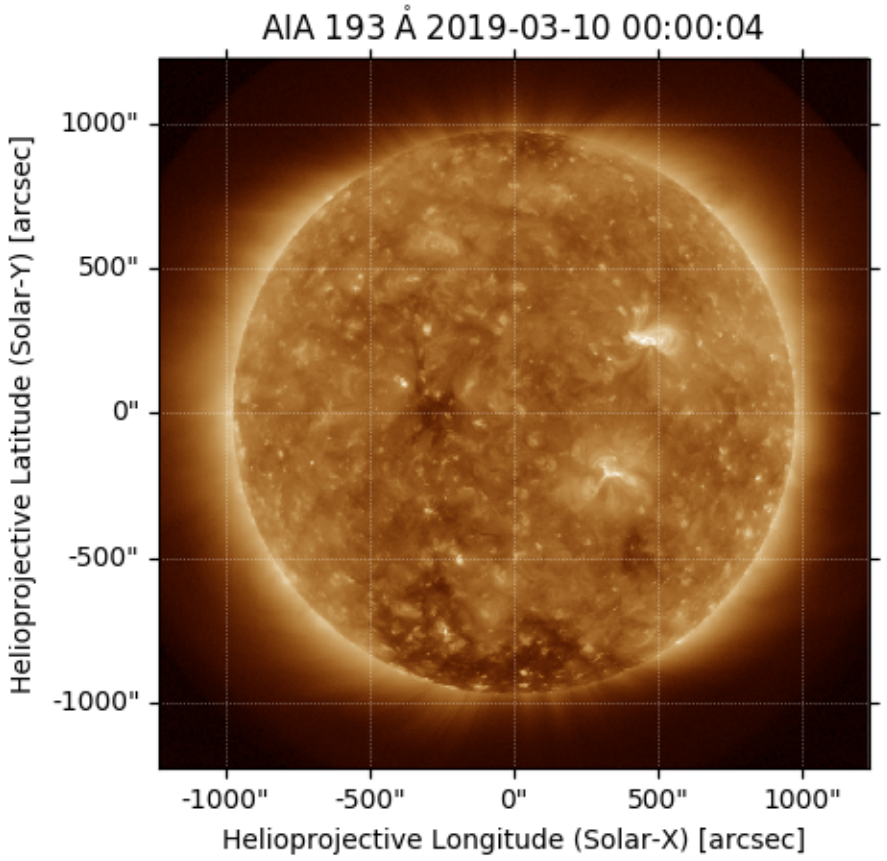


Out:

```
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 0x7f4d22cd50f0>
```

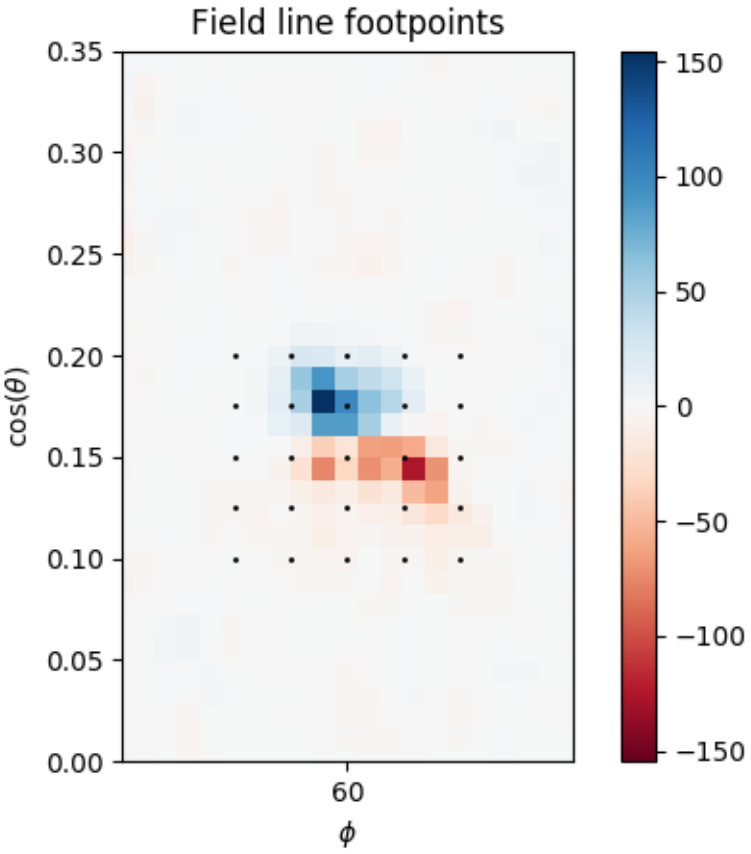
Now we construct a 10 x 10 grid of footpoitns to trace some magnetic field lines from.

The figure shows a zoom in of the magnetic field map, with the footpoints overplotted. The footpoints are centered around the active region metnioned above.

```
s, phi = np.meshgrid(np.linspace(0.1, 0.2, 5),
                     np.deg2rad(np.linspace(55, 65, 5)))

fig, ax = plt.subplots()
mesh = input.plot_input(ax)
fig.colorbar(mesh)
ax.scatter(np.rad2deg(phi), s, color='k', s=1)

ax.set_xlim(50, 70)
ax.set_ylim(0, 0.35)
ax.set_title('Field line footpoints')
```



Out:

```
Text(0.5, 1.0, 'Field line footpoints')
```

Compute the PFSS solution from the GONG magnetic field input

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

Trace field lines from the footpoints defined above. *pfsspy.coords* is used to convert the s, phi cooridnates into the cartesian coordinates that are needed by the tracer.

```
tracer = tracing.PythonTracer(atol=1e-6)
x0 = np.array(pfsspy.coords.strum2cart(0.01, s.ravel(), phi.ravel())).T
flines = tracer.trace(x0, output)
```

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

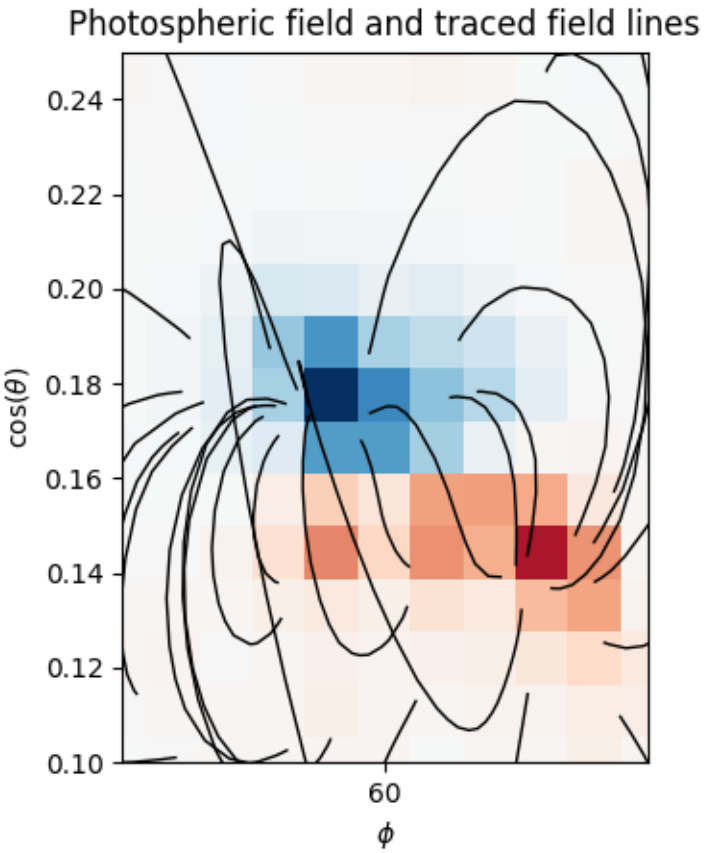
```
fig, ax = plt.subplots()
mesh = input.plot_input(ax)
for fline in flines:
    fline.coords.representation_type = 'spherical'
    ax.plot(fline.coords.lon / u.deg, np.sin(fline.coords.lat),
            color='black', linewidth=1)

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:

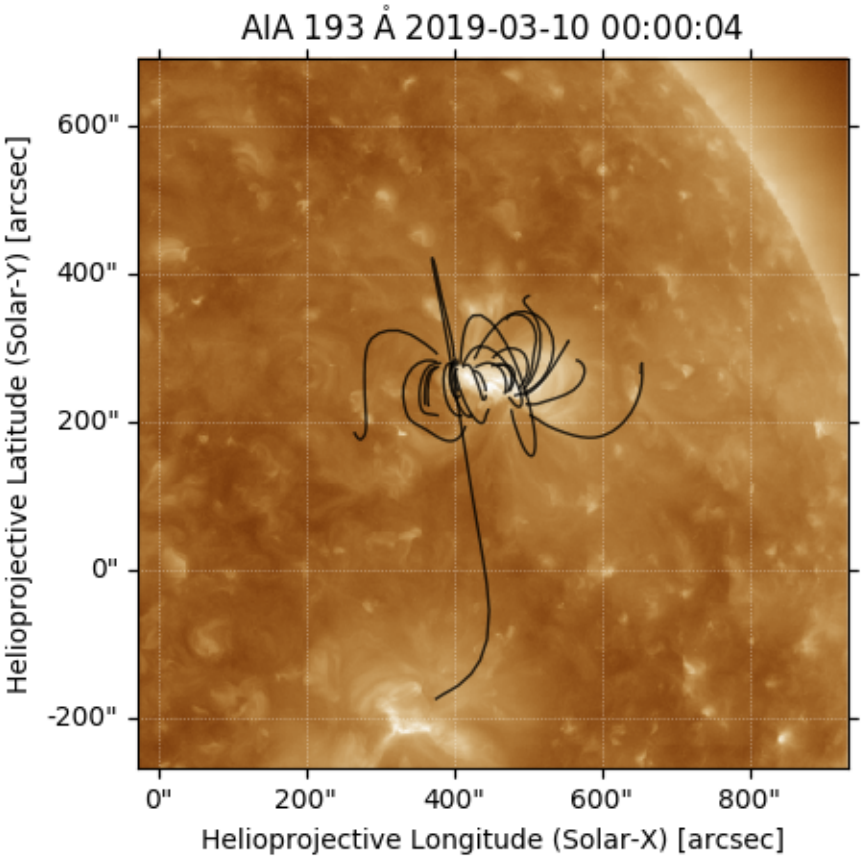
```
Text(0.5, 1.0, 'Photospheric field and traced field lines')
```

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:
    coords = fline.coords.transform_to(aia.coordinate_frame)
    ax.plot_coord(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 14.322 seconds)

and for the helper modules (behind the scense!) see

3.6 Helper modules

3.6.1 pfsspy.plot Module

Functions

contour(phi, costheta, field, levels[, ax])	
	Parameters
radial_cut(phi, costheta, field[, ax])	

contour

pfsspy.plot.**contour**(*phi*, *costheta*, *field*, *levels*, *ax=None*, ***kwargs*)

Parameters

phi :

costheta :

field :

levels :

ax [Axes, optional] Axes to plot to. If `None` a new figure is created.

****kwargs :** Keyword arguments are handed to *ax.contour*.

radial_cut

pfsspy.plot.**radial_cut**(*phi*, *costheta*, *field*, *ax=None*, ***kwargs*)

3.6.2 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

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.

cart2strum

pfsspy.coords.**cart2strum**(*x*, *y*, *z*)
Convert cartesian coordinates to strumfric coordinates.

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.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

pfsspy, 7
pfsspy.coords, 39
pfsspy.fieldline, 11
pfsspy.plot, 38
pfsspy.tracing, 14

INDEX

A

al (*pfsspy.Output attribute*), 10

B

bc (*pfsspy.Output attribute*), 10
bg (*pfsspy.Output attribute*), 10

C

cart2strum() (*in module pfsspy.coords*), 40
cartesian_to_coordinate() (*pfsspy.tracing.Tracer static method*), 16
closed_field_lines (*pfsspy.fieldline.FieldLines attribute*), 13
ClosedFieldLines (*class in pfsspy.fieldline*), 11
connectivities (*pfsspy.fieldline.FieldLines attribute*), 13
contour() (*in module pfsspy.plot*), 39

D

dp (*pfsspy.Grid attribute*), 8
dr (*pfsspy.Grid attribute*), 8
ds (*pfsspy.Grid attribute*), 8

E

expansion_factor (*pfsspy.fieldline.FieldLine attribute*), 12

F

FieldLine (*class in pfsspy.fieldline*), 12
FieldLines (*class in pfsspy.fieldline*), 13
FortranTracer (*class in pfsspy.tracing*), 15

G

Grid (*class in pfsspy*), 8

I

Input (*class in pfsspy*), 9
is_open (*pfsspy.fieldline.FieldLine attribute*), 12

L

load_output() (*in module pfsspy*), 7

O

open_field_lines (*pfsspy.fieldline.FieldLines attribute*), 13
OpenFieldLines (*class in pfsspy.fieldline*), 14
Output (*class in pfsspy*), 9

P

pc (*pfsspy.Grid attribute*), 8
pfss() (*in module pfsspy*), 7
pfsspy (*module*), 7
pfsspy.coords (*module*), 39
pfsspy.fieldline (*module*), 11
pfsspy.plot (*module*), 38
pfsspy.tracing (*module*), 14
pg (*pfsspy.Grid attribute*), 8
plot_input() (*pfsspy.Input method*), 9
plot_pil() (*pfsspy.Output method*), 10
plot_source_surface() (*pfsspy.Output method*), 10
polarities (*pfsspy.fieldline.FieldLines attribute*), 13
polarity (*pfsspy.fieldline.FieldLine attribute*), 12
PythonTracer (*class in pfsspy.tracing*), 15

R

radial_cut() (*in module pfsspy.plot*), 39
rc (*pfsspy.Grid attribute*), 8
rg (*pfsspy.Grid attribute*), 8

S

save() (*pfsspy.Output method*), 11
sc (*pfsspy.Grid attribute*), 8
sg (*pfsspy.Grid attribute*), 8
solar_feet (*pfsspy.fieldline.OpenFieldLines attribute*), 14
solar_footpoint (*pfsspy.fieldline.FieldLine attribute*), 12
source_surface_br (*pfsspy.Output attribute*), 10
source_surface_feet (*pfsspy.fieldline.OpenFieldLines attribute*), 14
source_surface_footpoint (*pfsspy.fieldline.FieldLine attribute*), 13

sph2cart() (*in module pfsspy.coords*), 40
strum2cart() (*in module pfsspy.coords*), 40

T

trace() (*pfsspy.Output method*), 11
trace() (*pfsspy.tracing.FortranTracer method*), 15
trace() (*pfsspy.tracing.PythonTracer method*), 16
trace() (*pfsspy.tracing.Tracer method*), 16
Tracer (*class in pfsspy.tracing*), 16

V

validate_seeds_shape() (*pfsspy.tracing.Tracer static method*), 16