Pleque Documentation

Release 0.0.5

Lukas Kripner

Jan 26, 2021

Contents:

Code home: https://github.com/kripnerl/pleque/

PLEQUE is a code which allows easy and quick access to tokamak equilibria obtained by solving the Grad-Shafranov equation. To get started, see the *First steps* and the *Examples*. The code is produced at the Institute of Plasma Physics of the Czech Academy of Sciences, Prague, by Lukáš Kripner (kripner@ipp.cas.cz) and his colleagues.

CHAPTER 1

First steps

1.1 Prerequisites

The following packages are required to install PLEQUE:

```
python>=3.5
numpy
scipy
shapely
scikit-image
xarray
pandas
h5py
omas
```

They should be automatically handled by pip further in the installation process.

1.2 Download the source code and install PLEQUE

First, pick where you wish to install the code:

```
cd /desired/path/
```

There are two options how to get the code: from PyPI or by cloning the repository.

1.2.1 Install from PyPI (https://pypi.org/project/pleque/)

```
pip install --user pleque
```

Alternatively, you may use the unstable experimental release (probably with more fixed bugs):

```
pip install --user -i https://test.pypi.org/simple/ pleque
```

1.2.2 Install after cloning the github repository

```
git clone https://github.com/kripnerl/pleque.git
cd pleque
pip install --user .
```

Congratulations, you have installed PLEQUE!

CHAPTER 2

Examples

Browse examples of using PLEQUE with these **Jupyter notebooks**:

2.1 Basic example

The following example shows how to load an equilibrium saved in the EQDSK format and perform some basic operations with it. Several test equilibria come shipped with PLEQUE; here we will use onee of them.

[1]:
```
from pleque.io import readers
import pkg_resources
import matplotlib as plt

#Locate the test equilibrium
filepath = pkg_resources.resource_filename('pleque', 'resources/baseline_eqdsk')
```

The heart of PLEQUE is its `Equilibrium` class, which contains all the equilibrium information (and much more). Typically its instances are called `eq`.

[2]:
```
# Create an instance of the `Equilibrium` class
eq = readers.read_geqdsk(filepath)
```

```
  nx = 65, ny = 129
197 1
```

The `Equilibrium` class comes with many interesting functions and caveats.

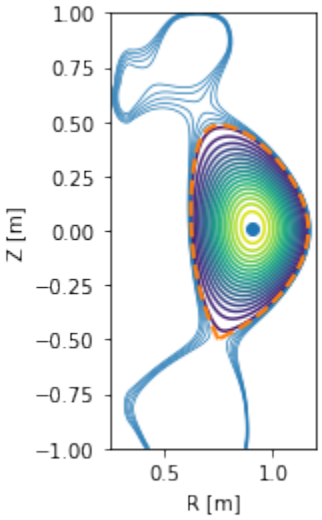
[3]:
```
# Plot a simple overview of the equilibrium
eq.plot_overview()

# Calculate the separatrix area
sep_area = eq.lcfs.area
print('Separatrix area: A_sep = %.3f m^2' % sep_area)
```

(continues on next page)

(continued from previous page)

```
# Get absolute magnetic field magnitude at given point
R = 0.7 #m
Z = 0.1 #m
B = eq.B_abs(R, Z)
print('Magnetic field at R=%.1f m and Z=%.1f m: B = %.1f T' % (R, Z, B))
```

```
Separatrix area: A_sep = 0.381 m^2
Magnetic field at R=0.7 m and Z=0.1 m: B = 6.7 T
```



Browse various attributes and functions of the `Equilibrium` class to see what it has to offer.

2.2 PLEQUE vs raw reconstruction

In this notebook, we demonstrate that PLEQUE is better than raw reconstruction at everything.

[1]:
```
%pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

[2]:
```
from pleque.io import _geqdsk as eqdsktool
from pleque.io.readers import read_geqdsk
from pleque.utils.plotting import *
#from pleque import Equilibrium
from pleque.tests.utils import get_test_equilibria_filenames, load_testing_equilibrium
```

2.2.1 Load a testing equilibrium

Several test equilibria come shipped with PLEQUE. Their location is:

[3]:
```
gfiles = get_test_equilibria_filenames()
gfiles
```

[3]:
```
['/home/docs/checkouts/readthedocs.org/user_builds/pleque/envs/unstable/lib/python3.6/
↪site-packages/pleque/resources/baseline_eqdsk',
 '/home/docs/checkouts/readthedocs.org/user_builds/pleque/envs/unstable/lib/python3.6/
↪site-packages/pleque/resources/scenario_1_baseline_upward_eqdsk',
```

(continues on next page)

(continued from previous page)

```
 '/home/docs/checkouts/readthedocs.org/user_builds/pleque/envs/unstable/lib/python3.6/
↪site-packages/pleque/resources/DoubleNull_eqdsk',
 '/home/docs/checkouts/readthedocs.org/user_builds/pleque/envs/unstable/lib/python3.6/
↪site-packages/pleque/resources/g13127.1050',
 '/home/docs/checkouts/readthedocs.org/user_builds/pleque/envs/unstable/lib/python3.6/
↪site-packages/pleque/resources/14068@1130_2kA_modified_triang.gfile',
 '/home/docs/checkouts/readthedocs.org/user_builds/pleque/envs/unstable/lib/python3.6/
↪site-packages/pleque/resources/g15349.1120',
 '/home/docs/checkouts/readthedocs.org/user_builds/pleque/envs/unstable/lib/python3.6/
↪site-packages/pleque/resources/shot8078_jorek_data.nc']
```

Load the equilibrium directly

Here the test equilibrium is directly loaded and stored in the variable `eq_efit`. The variable then contains all equilibrium information calculated by EFIT in the form of a dictionary.

[4]:
```
test_case_number = 5

with open(gfiles[test_case_number], 'r') as f:
    eq_efit = eqdsktool.read(f)
eq_efit.keys()
```

```
  nx = 33, ny = 33
361 231
```

[4]:
```
dict_keys(['nx', 'ny', 'rdim', 'zdim', 'rcentr', 'rleft', 'zmid', 'rmagx', 'zmagx',
↪'simagx', 'sibdry', 'bcentr', 'cpasma', 'F', 'pres', 'FFprime', 'pprime', 'psi', 'q
↪', 'rbdry', 'zbdry', 'rlim', 'zlim'])
```

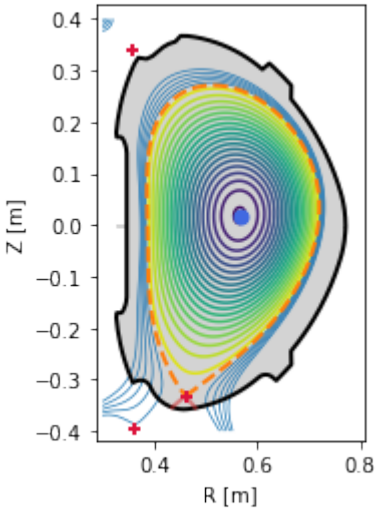
Load equilibrium using PLEQUE

PLEQUE loads the same file at its core, but it wraps it in the `Equilibrium` class and stores it in the variable `eq`.

[5]:
```
def save_it(*args,**kwargs):
    pass
```

[6]:
```
#Load equilibrium stored in the EQDSK format
eq = read_geqdsk(gfiles[test_case_number])

#Plot basic overview of the equilibrium
plt.figure()
eq._plot_overview()

#Plot X-points
plot_extremes(eq, markeredgewidth=2)
```

```
  nx = 33, ny = 33
361 231
```



2.2.2 PLEQUE vs raw reconstruction: spatial resolution near the X-point

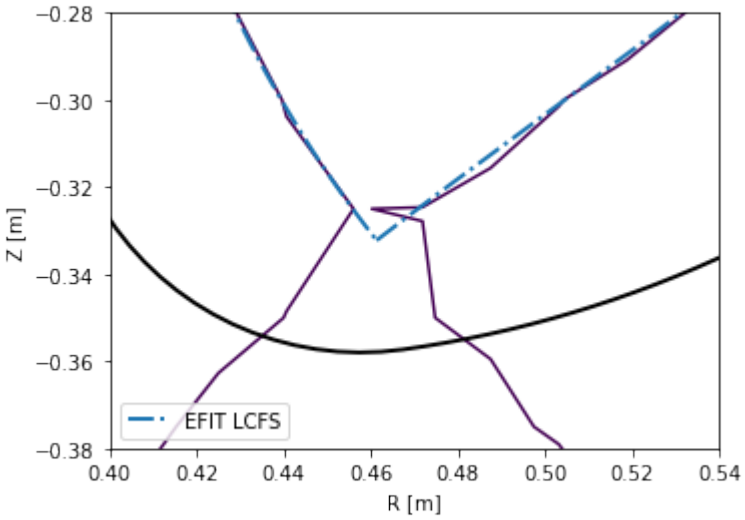
EFIT output (Ψ, j etc.) is given on a rectangular grid:

[7]:
```
r_axis = np.linspace(eq_efit["rleft"], eq_efit["rleft"] + eq_efit["rdim"], eq_efit["nx
↪"])
z_axis = np.linspace(eq_efit["zmid"] - eq_efit["zdim"] / 2, eq_efit["zmid"] + eq_efit[
↪"zdim"] / 2, eq_efit["ny"])
```

To limit the file size, the grid has a finite resolution. This means that in areas where high spatial resolution is needed (for instance the X-point vicinity), raw reconstructions are usually insufficient. The following figure demonstrates this.

[8]:
```
plt.figure()
ax = plt.gca()

#Limiter (stored in EFIT output)
ax.plot(eq_efit['rlim'], eq_efit['zlim'], color='k', lw=2)

#Magnetic surface defined by Psi == eq_efit['sibdry']
ax.contour(r_axis, z_axis, eq_efit['psi'].T, [eq_efit['sibdry']])

#Magnetic surface saved as the LCFS in EFIT output
ax.plot(eq_efit['rbdry'], eq_efit['zbdry'], 'C0-.', lw=2, label='EFIT LCFS')

ax.set_xlabel('R [m]')
ax.set_ylabel('Z [m]')
ax.set_aspect('equal')
plt.legend()
ax.set_xlim(0.4, 0.54)
ax.set_ylim(-0.38, -0.28)
```

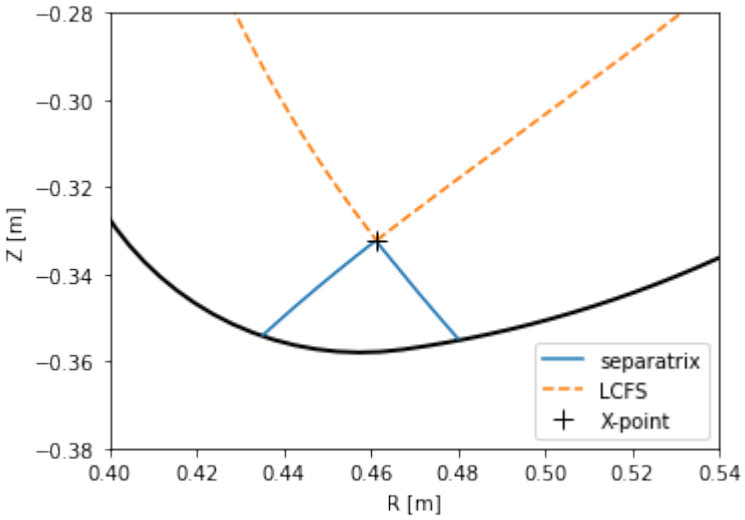
[8]:
```
(-0.38, -0.28)
```



PLEQUE, however, performs equilibrium interpolation that can easily produce the same plots in a much higher spatial resolution.

[9]:
```
plt.figure()
ax = plt.gca()

#Limiter (accessed through the Equilibrium class)
eq.first_wall.plot(ls="-", color="k", lw=2)

#Separatrix, cropped to its part inside the first wall
inside_fw = eq.in_first_wall(eq.separatrix)
separatrix = eq.coordinates(R=eq.separatrix.R[inside_fw], Z=eq.separatrix.Z[inside_
↪fw])
separatrix.plot(label='separatrix')

#LCFS (without strike points)
eq.lcfs.plot(color='C1', ls='--', label='LCFS')

#X-point
ax.plot(eq._x_point[0], eq._x_point[1], 'k+', markersize=10, label='X-point')

ax.set_xlabel('R [m]')
ax.set_ylabel('Z [m]')
ax.set_aspect('equal')
plt.legend()
ax.set_xlim(0.4, 0.54)
ax.set_ylim(-0.38, -0.28)
```

[9]:
```
(-0.38, -0.28)
```



2.2.3 PLEQUE vs raw reconstruction: q profile

The safety factor q can be defined as the number of toroidal turns a magnetic field line makes along its magnetic surface before it makes a full poloidal turn. Since the poloidal field is zero at the X-point, the magnetic field lines inside the separatrix are caught in an infinite toroidal loop at the X-point and $q \to +\infty$. (This is why the edge safety factor is given as q_{95} at $\psi_N = 0.95$. If it were given an $\psi_N = 1.00$, its value would diverge regardless of its profile shape.)

In this section we compare several methods of calculating q:

1. q as calculated by the reconstruction itself (`q_efit`)
2. q evaluated by `eq.q(q_eq)`
3. q evaluated by `eq._flux_surface(psi_n).eval_q`
   - using the default, rectangle rule (`q1`)
   - using the trapezoidal rule (`q2`)
   - using the Simpson rule (`q3`)

Method 3 calculates the safety factor according to formula (5.35) in [Jardin, 2010: Computation Methods in Plasma Physics]:

$$q(\psi) = \frac{gV'}{(2\pi)^2\Psi'}\langle R^{-2}\rangle$$

where V' is the differential volume and, in PLEQUE's notation, $g(\psi) \equiv F(\psi)$ and $\Psi \equiv \psi$ (and therefore $\Psi' \equiv d\Psi/d\psi = 1$). Furthermore, the surface average $\langle\cdot\rangle$ of an arbitrary function a is defined as $\langle a\rangle = \frac{2\pi}{V'}\int_0^{2\pi} d\theta J a$ where J is the Jacobian. Putting everything together, one obtains the formula used by PLEQUE:

$$q(\psi) = \frac{F(\psi)}{2\pi}\int_0^{2\pi} d\theta J R^{-2}$$

where, based on the convention defined by COCOS, the factor 2π can be missing and q may be either positive or negative. (In the default convention of EFIT, COCOS 3, q is negative.) Finally, the integral can be calculated with three different methods: the rectangle rule (resulting in `q1`), the trapezoidal rule (resulting in `q2`) and the Simpson rule (resulting in `q3`).

Method 2 is based on method 3. The safety factor profile is calculated for 200 points in $\psi_N \in (0, 1)$ and interpolated with a spline. `eq.q` then invokes this spline to calculate q at any given ψ_N.

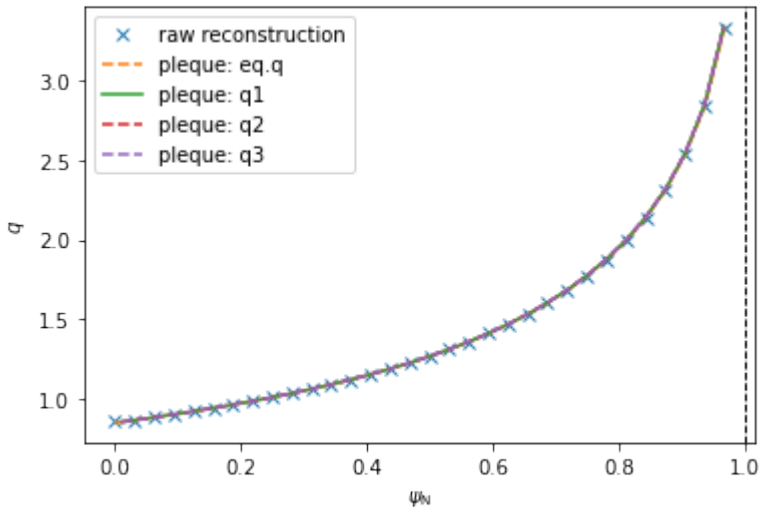
[10]:
```
#q taken directly from the reconstruction
q_efit = eq_efit['q']
q_efit = q_efit[:-1] #in some reconstructions, q is calculated up to psi_N=1
psi_efit = np.linspace(0, 1, len(q_efit), endpoint=False)
#psi_efit2 = np.linspace(0, 1, len(q_efit), endpoint=True)
# If you try this for several test equilibria, you will find that some give q at Psi_
↪N=1
# and some stop right short of Psi_N=1. To test which is which, try both including and
# excluding the endpoint in the linspace definition.

#q stored in the Equilibrium class
coords = eq.coordinates(psi_n = np.linspace(0, 1, len(q_efit), endpoint=False))
psi_eq = coords.psi_n
q_eq = abs(eq.q(coords))

#q calculated by eq._flux_surface(Psi).eval_q
surf_psin = linspace(0.01, 1, len(q_efit), endpoint=False)
surfs = [eq._flux_surface(psi_n=psi_n)[0] for psi_n in surf_psin]
surf_psin = [np.mean(s.psi_n) for s in surfs]
q1 = abs(np.array([np.asscalar(s.eval_q) for s in surfs]))
q2 = abs(np.array([np.asscalar(s.get_eval_q('trapz')) for s in surfs]))
q3 = abs(np.array([np.asscalar(s.get_eval_q('simps')) for s in surfs]))
```

```
/home/docs/checkouts/readthedocs.org/user_builds/pleque/envs/unstable/lib/python3.6/
↪site-packages/ipykernel_launcher.py:19: DeprecationWarning: np.asscalar(a) is␣
↪deprecated since NumPy v1.16, use a.item() instead
/home/docs/checkouts/readthedocs.org/user_builds/pleque/envs/unstable/lib/python3.6/
↪site-packages/ipykernel_launcher.py:20: DeprecationWarning: np.asscalar(a) is␣
↪deprecated since NumPy v1.16, use a.item() instead
/home/docs/checkouts/readthedocs.org/user_builds/pleque/envs/unstable/lib/python3.6/
↪site-packages/ipykernel_launcher.py:21: DeprecationWarning: np.asscalar(a) is␣
↪deprecated since NumPy v1.16, use a.item() instead
```

Notice the absolute value; this is required because $q < 0$ in the convention used here.

[11]:
```
#q profile comparison
plt.figure()
plt.plot(psi_efit, q_efit, 'x', label='raw reconstruction')
#plt.plot(psi_efit2, q_efit, 'x', label='raw reconstruction')
plt.plot(psi_eq, q_eq, '--', label=r'pleque: eq.q')
plt.plot(surf_psin, q1, '-', label=r'pleque: q1')
plt.plot(surf_psin, q2, '--', label=r'pleque: q2')
plt.plot(surf_psin, q3, '--', label=r'pleque: q3')
plt.xlabel(r'$\psi_\mathrm{N}$')
plt.ylabel(r'$q$')
plt.axvline(1, ls='--', color='k', lw=1)
plt.legend()
```

[11]:
```
<matplotlib.legend.Legend at 0x7f0dd17e96a0>
```



Investigating the differences between the five q profiles shows quite a good agreement. The profiles disagree slightly near $\psi_N \to 0$ since the safety factor is defined by a limit here. (Notice that, using method 3, the ψ_N axis begins at 0.01 and not 0. This is because q cannot be calculated by the formula above in $\psi_N = 0$ and the algorithm fails.)

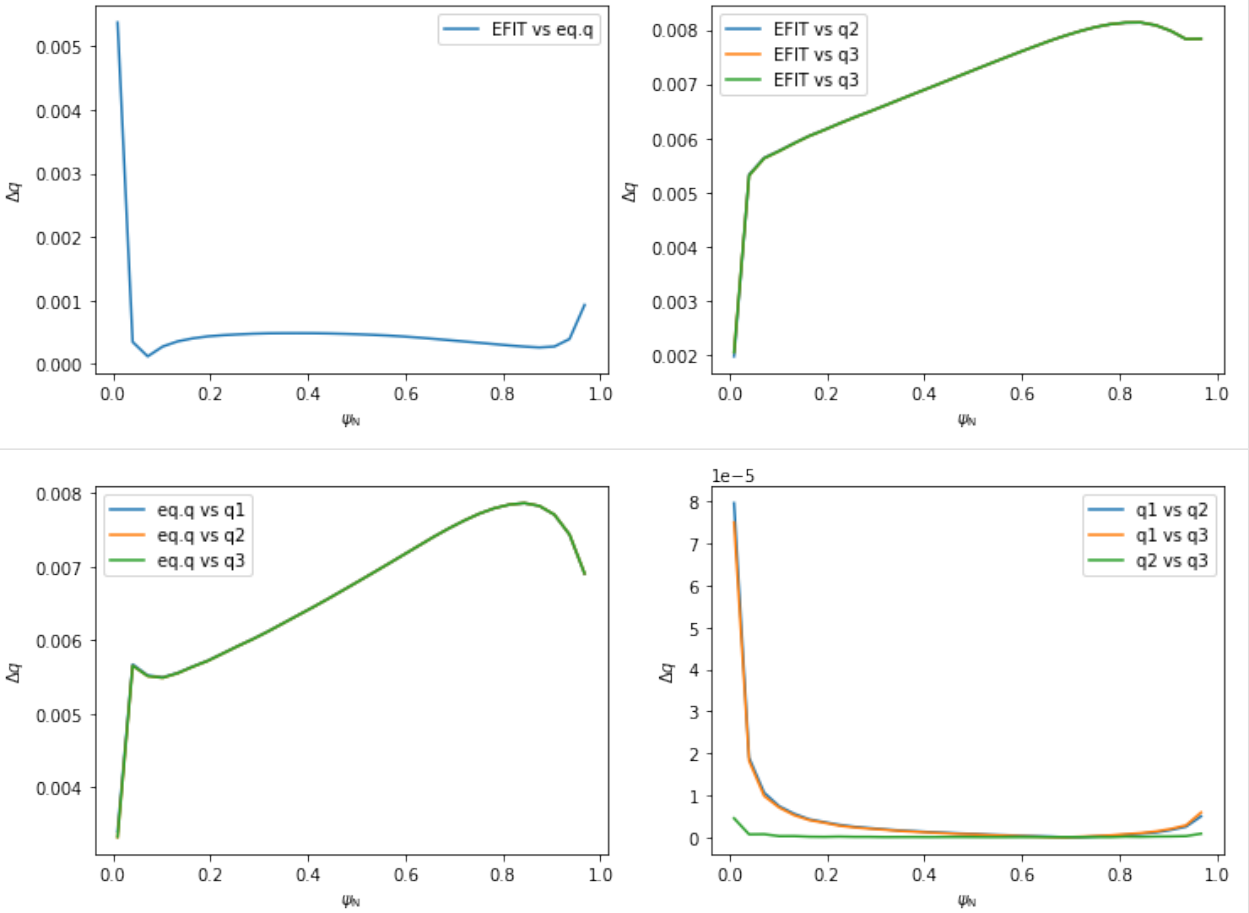
[12]:
```
plt.figure(figsize=(12,4))

#EFIT vs eq.q
plt.subplot(121)
plt.plot(surf_psin, abs(q_eq-q_efit), label='EFIT vs eq.q')
plt.legend()
plt.xlabel(r'$\psi_\mathrm{N}$')
plt.ylabel(r'$\Delta q$')

#EFIT vs q1-q3
plt.subplot(122)
plt.plot(surf_psin, abs(q_efit-q1), label='EFIT vs q2')
plt.plot(surf_psin, abs(q_efit-q2), label='EFIT vs q3')
plt.plot(surf_psin, abs(q_efit-q3), label='EFIT vs q3')
plt.legend()
plt.xlabel(r'$\psi_\mathrm{N}$')
plt.ylabel(r'$\Delta q$')


plt.figure(figsize=(12,4))

#eq.q vs all the rest
plt.subplot(121)
plt.plot(surf_psin, abs(q_eq-q1), label='eq.q vs q1')
plt.plot(surf_psin, abs(q_eq-q2), label='eq.q vs q2')
plt.plot(surf_psin, abs(q_eq-q3), label='eq.q vs q3')
plt.legend()
plt.xlabel(r'$\psi_\mathrm{N}$')
plt.ylabel(r'$\Delta q$')

#q1 vs q2 vs q3
plt.subplot(122)
plt.plot(surf_psin, abs(q1-q2), label='q1 vs q2')
plt.plot(surf_psin, abs(q1-q3), label='q1 vs q3')
```

(continues on next page)

(continued from previous page)

```
plt.plot(surf_psin, abs(q2-q3), label='q2 vs q3')
plt.legend()
plt.xlabel(r'$\psi_\mathrm{N}$')
plt.ylabel(r'$\Delta q$')
```

[12]:
```
Text(0, 0.5, '$\\Delta q$')
```



2.2.4 Plotting contour plots of various quantities

In this section PLEQUE is used to produce contour plots of the following quantities:

- poloidal magnetic field flux ψ
- toroidal magnetic field flux
- poloidal magnetic field B_p
- toroidal magnetic field B_t
- total magnetic field $|B|$
- total pressure p
- toroidal current density j_ϕ
- poloidal current density j_θ

First, a general plotting function `plot_2d` is defined.

[13]:
```
def plot_2d(R, Z, data, *args, title=None):

    #Define X and Y axis limits based on the vessel size
    rlim = [np.min(eq.first_wall.R), np.max(eq.first_wall.R)]
    zlim = [np.min(eq.first_wall.Z), np.max(eq.first_wall.Z)]
    size = rlim[1] - rlim[0]
    rlim[0] -= size / 12
    rlim[1] += size / 12
    size = zlim[1] - zlim[0]
    zlim[0] -= size / 12
    zlim[1] += size / 12

    #Set up the figure: set axis limits, draw LCFS and first wall, write labels
    ax = plt.gca()
    ax.set_xlim(rlim)
    ax.set_ylim(zlim)
    ax.plot(eq.lcfs.R, eq.lcfs.Z, color='k', ls='--', lw=2)
    ax.plot(eq.first_wall.R, eq.first_wall.Z, 'k-', lw=2)
    ax.set_xlabel('R [m]')
    ax.set_ylabel('Z [m]')
    ax.set_aspect('equal')
    if title is not None:
        ax.set_title(title)

    #Finally, plot the desired quantity
    cl = ax.contour(R, Z, data, *args)

    return cl
```

Now we set up an $[R, Z]$ grid where these quantities are evaluated and plot the quantities.

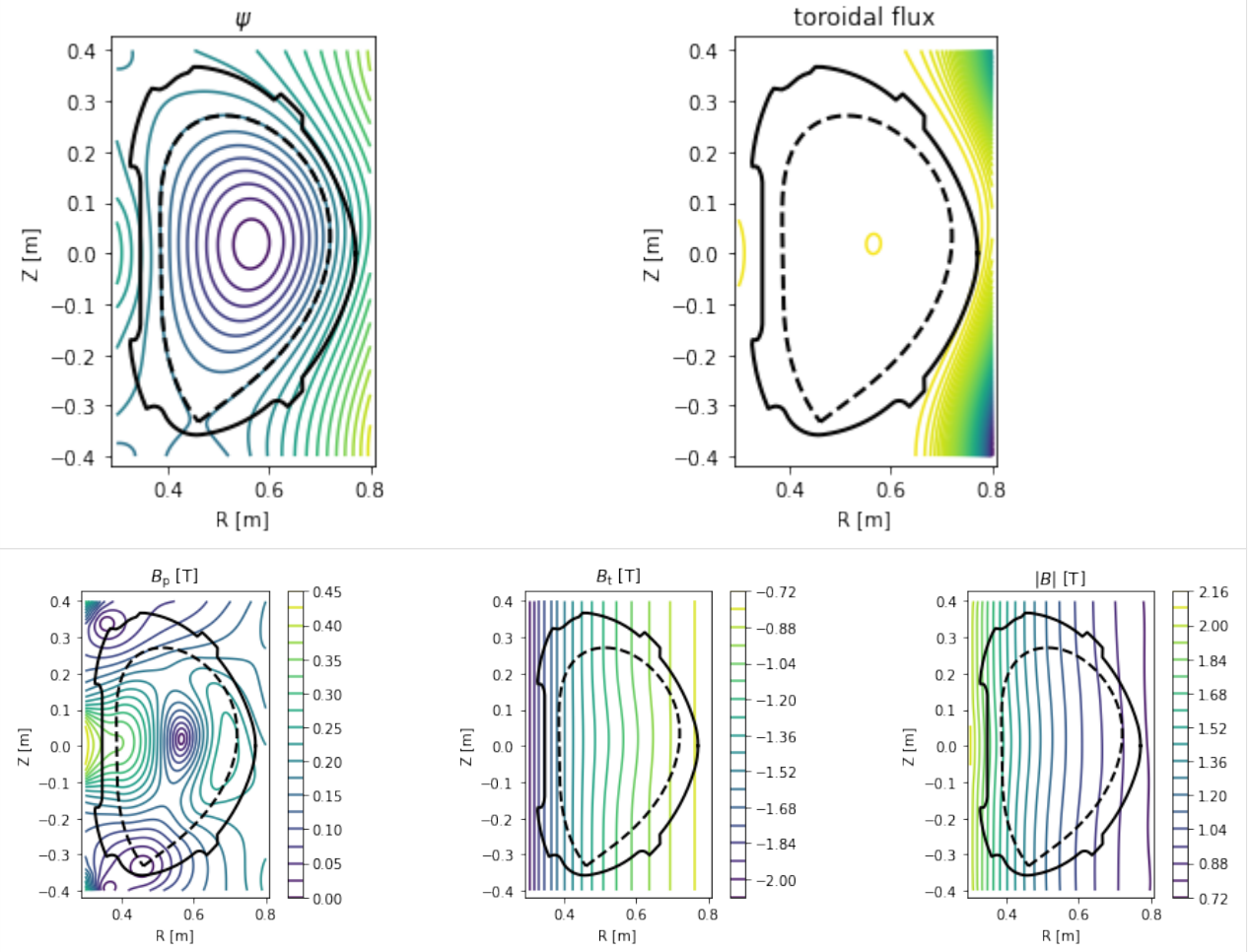
[14]:
```
#Create an [R,Z] grid 200 by 300 points
grid = eq.grid((200,300), dim='size')

#Plot the poloidal flux and toroidal flux
plt.figure(figsize=(16,4))
plt.subplot(131)
plot_2d(grid.R, grid.Z, grid.psi, 20, title=r'$\psi$')
plt.subplot(132)
plot_2d(grid.R, grid.Z, eq.tor_flux(grid), 100, title='toroidal flux')

#Plot the poloidal magnetic field, toroidal magnetic field and the total magnetic␣
↪field
plt.figure(figsize=(16,4))
plt.subplot(131)
cl = plot_2d(grid.R, grid.Z, eq.B_pol(grid), 20, title=r'$B_\mathrm{p}$ [T]')
plt.colorbar(cl)
plt.subplot(132)
cl = plot_2d(grid.R, grid.Z, eq.B_tor(grid), 20, title=r'$B_\mathrm{t}$ [T]')
plt.colorbar(cl)
plt.subplot(133)
cl = plot_2d(grid.R, grid.Z, eq.B_abs(grid), 20, title=r'$|B|$ [T]')
plt.colorbar(cl)

#Plot the total pressure, toroidal current density and poloidal current density
plt.figure(figsize=(16,4))
```

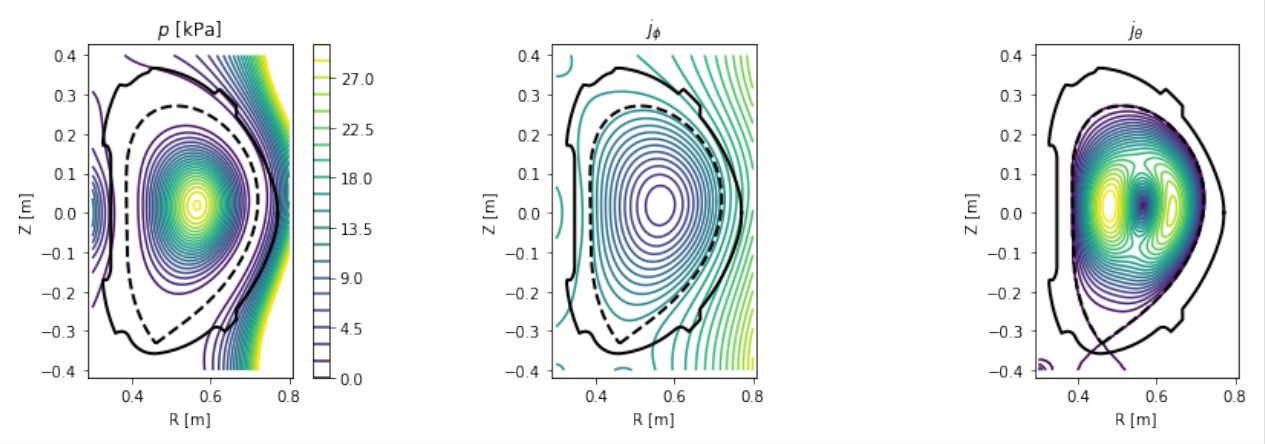
(continues on next page)

(continued from previous page)

```
plt.subplot(131)
cl = plot_2d(grid.R, grid.Z, eq.pressure(grid)/1e3, np.linspace(0, 30, 21), title=r'$p
↪$ [kPa]')
plt.colorbar(cl)
plt.subplot(132)
plot_2d(grid.R, grid.Z, eq.j_tor(grid), np.linspace(-5e6, 5e6, 30), title=r'$j_\phi$')
plt.subplot(133)
plot_2d(grid.R, grid.Z, eq.j_pol(grid),  np.linspace(0, 3e5, 21), title=r'$j_\theta$')
```

[14]:
```
<matplotlib.contour.QuadContourSet at 0x7f0dd0e76550>
```



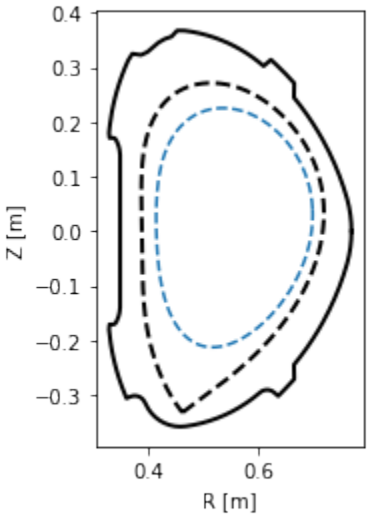


2.2.5 Exploring flux surface properties

With the `eq._flux_surface(psi_n)` function, one may study individual flux surfaces. In this section, we plot the $\psi_N = 0.8$ flux surface and calculate its safety factor q, length in the poloidal direction, total 3D area, volume and toroidal current density.

[15]:
```
#Define the flux surface by its normalised poloidal flux
surf = eq._flux_surface(psi_n=0.8)[0]

#Plot the flux surface
plt.figure()
ax = gca()
ax.plot(eq.lcfs.R, eq.lcfs.Z, color='k', ls='--', lw=2)
ax.plot(eq.first_wall.R, eq.first_wall.Z, 'k-', lw=2)
surf.plot(ls='--')
ax.set_xlabel('R [m]')
ax.set_ylabel('Z [m]')
ax.set_aspect('equal')

#Calculate several flux surface quantities
print('Safety factor: %.2f' % surf.eval_q[0])
print('Length: %.2f m' % surf.length)
print('Area: %.4f m^2' % surf.area)
print('Volume: %.3f m^3' % surf.volume)
print('Toroidal current density: %.3f MA/m^2' % (surf.tor_current/1e6))
```

```
Safety factor: -1.94
Length: 1.16 m
Area: 0.0989 m^2
Volume: 0.339 m^3
Toroidal current density: -0.213 MA/m^2
```



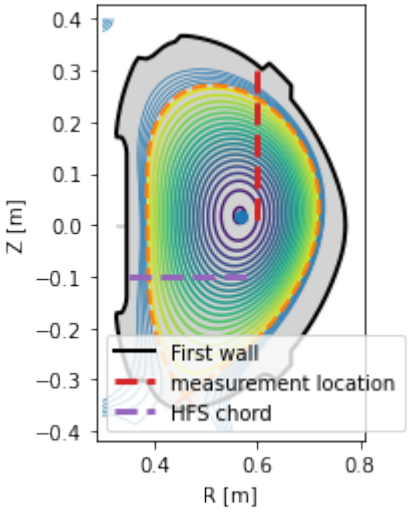
2.2.6 Profile mapping

In experiment one often encounters the need to compare profiles which were measured at various locations in the tokamak. In this section, we show how such a profile may be mapped onto an arbitrary location and to the outer midplane.

The profile is measured at the plasma top (in red) and mapped to the HFS (in violet) and the outer midplane (not shown).

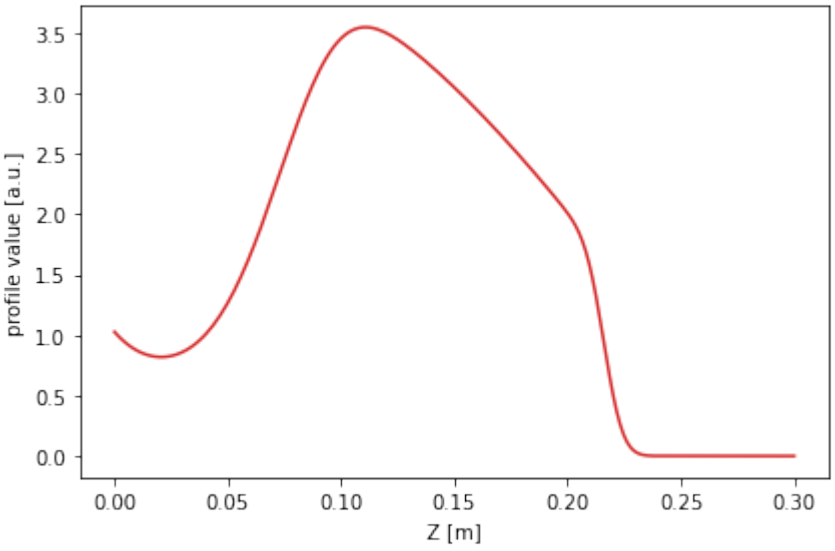
[16]:
```
#Define the chord along which the profile was measured (in red)
N = 200 #number of datapoints in the profile
chord = eq.coordinates(R=0.6*np.ones(N), Z=np.linspace(0.3, 0., N))

#Define the HFS chord where we wish to map the profile (in violet)
chord_hfs = eq.coordinates(R=np.linspace(0.35, 0.6, 20), Z=-0.1*np.ones(20))

#Plot both the chords
plt.figure()
eq._plot_overview()
chord.plot(lw=3, ls='--', color='C3', label='measurement location')
chord_hfs.plot(lw=3, ls='--', color='C4', label='HFS chord')
plt.legend(loc=3)
```

[16]:
```
<matplotlib.legend.Legend at 0x7f0dd0ed0240>
```



The profile shape is defined using the error function `erf`.

[17]:
```
from scipy.special import erf

#Define the profile values
prof_func = lambda x, k1, xsep: k1/4 * (1 + erf((x-xsep)*20))*np.log((x+1)*1.2) -␣
↪4*np.exp(-(50*(x-1)**2))
profile = prof_func(1 - chord.psi_n, 10, 0.15)

#Plot the profile along the chord it was measured at
plt.figure()
plt.plot(chord.Z, profile, color='C3')
plt.xlabel('Z [m]')
plt.ylabel('profile value [a.u.]')
plt.tight_layout()
```



To begin the mapping, the profile is converted into a flux function by `eq.fluxfuncs.add_flux_func()`. The flux function is a spline, and therefore it can be evaluated at any ψ_N coordinate covered by the original chord. This will allow its mapping to any other coordinate along the flux surfaces.

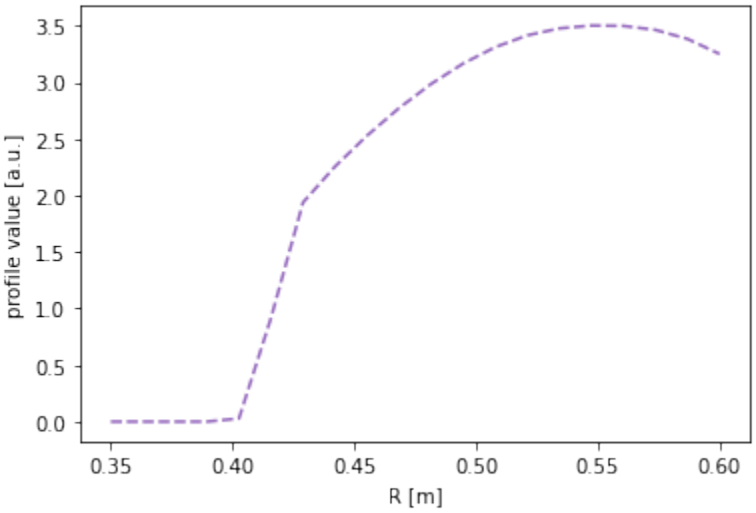
[18]:
```
eq.fluxfuncs.add_flux_func('test_profile', profile, chord, spline_smooth=0)
```

[18]:
```
<scipy.interpolate.fitpack2.InterpolatedUnivariateSpline at 0x7f0dd100d0b8>
```

To evaluate the flux function along a chord, simply pass the chord (an instance of the `Coordinates` class) to the flux function. In the next figure the profile is mapped to the HFS cord.

[19]:
```
#Map the profile to the HFS cord
plt.figure()
plt.plot(chord_hfs.R, eq.fluxfuncs.test_profile(chord_hfs), '--', color='C4')
plt.xlabel('R [m]')
plt.ylabel('profile value [a.u.]')
```

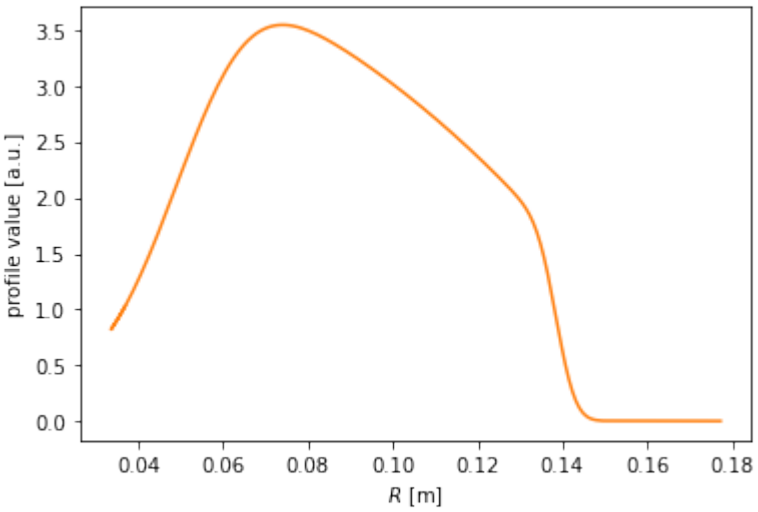
[19]:
```
Text(0, 0.5, 'profile value [a.u.]')
```



For the outer midplane, no special chord need be specified. Every instance of the `Coordinates` class can automatically map its coordinates to the outer midplane. (Note that this doesn't require a flux function to be specified. The conversion is performed in the coordinates only.)

[20]:
```
#Map the profile to the outer midplane
plt.figure()
plt.plot(chord.r_mid, profile, color='C1')
plt.xlabel(r'$R$ [m]')
plt.ylabel('profile value [a.u.]')
```

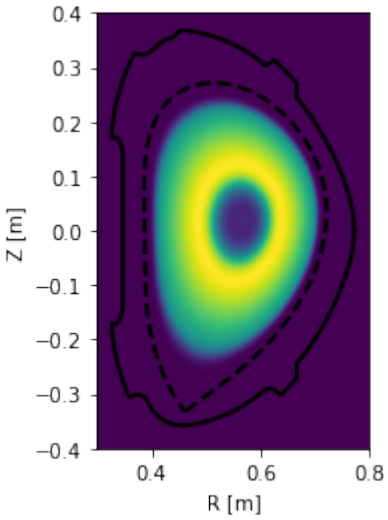
[20]:
```
Text(0, 0.5, 'profile value [a.u.]')
```



Finally, the profile may be drawn along the entire poloidal cross section.

[21]:
```
#Assuming poloidal symmetry, plot the profile in the poloidal cross section
plt.figure()
ax = gca()
ax.plot(eq.lcfs.R, eq.lcfs.Z, color='k', ls='--', lw=2)
ax.plot(eq.first_wall.R, eq.first_wall.Z, 'k-', lw=2)
grid = eq.grid()
ax.pcolormesh(grid.R, grid.Z, eq.fluxfuncs.test_profile(grid))
ax.set_xlabel('R [m]')
ax.set_ylabel('Z [m]')
ax.set_aspect('equal')
```

```
/home/docs/checkouts/readthedocs.org/user_builds/pleque/envs/unstable/lib/python3.6/
↪site-packages/ipykernel_launcher.py:7: MatplotlibDeprecationWarning: shading='flat'␣
↪when X and Y have the same dimensions as C is deprecated since 3.3.  Either specify␣
↪the corners of the quadrilaterals with X and Y, or pass shading='auto', 'nearest'␣
↪or 'gouraud', or set rcParams['pcolor.shading'].  This will become an error two␣
↪minor releases later.
  import sys
```



2.2.7 Detector line of sight visualisation

In this section, we demonstrate the flexibility of the `Coordinates` class by visualising a detector line of sight. Suppose we have a pixel detector at the position $[X, Y, Z] = [1.2\,\mathrm{m}, 0\,\mathrm{m}, -0.1\,\mathrm{m}]$.

[22]:
```
# Define detector position [X, Y, Z]
position = np.array((1.2, 0, -0.1))
```

The detector views the plasma mostly tangentially to the toroidal direction, but also sloping a little upward.

[23]:
```
#Define the line of sight direction (again along [X, Y, Z])
direction = np.array((-1, 0.6, 0.2))

#Norm the direction to unit length
direction /= np.linalg.norm(direction)
```

Now since the plasma geometry is curvilinear, the detector line of sight is not trivial. Luckily PLEQUE's `Coordinates` class can easily express its stored coordinates both in the cartesian $[X, Y, Z]$ and the cylindrical $[R, Z, \phi]$ coordinate systems. In the following line, 20 points along the detector line of sight are calculated in 3D.

[24]:
```
# Calculate detector line of sight (LOS)
LOS = eq.coordinates(position + direction[np.newaxis,:] * np.linspace(0, 2.0, 20)[:,␣
↪np.newaxis],
                     coord_type=('X', 'Y', 'Z')
                    )
```

To visualise the line of sight in top view $[X, Y]$ and poloidal cross-section view $[R, Z]$, we first define the limiter outline as viewed from the top. Then we proceed with the plotting.

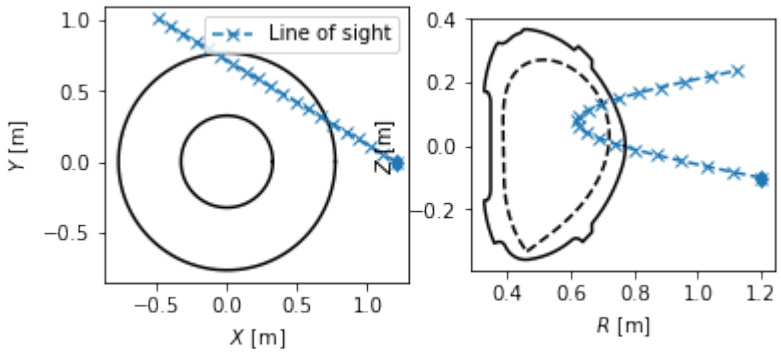
[25]:
```
# Limiter outline viewed from the top
Ns = 100
inner_lim = eq.coordinates(np.min(eq.first_wall.R)*np.ones(Ns), np.zeros(Ns), np.
↪linspace(0, 2*np.pi, Ns))
outer_lim = eq.coordinates(np.max(eq.first_wall.R)*np.ones(Ns), np.zeros(Ns), np.
↪linspace(0, 2*np.pi, Ns))

# Prepare figure
fig, axs = plt.subplots(1,2)

# Top view
ax = axs[0]
ax.plot(inner_lim.X, inner_lim.Y, 'k-')
ax.plot(outer_lim.X, outer_lim.Y, 'k-')
ax.plot(LOS.X, LOS.Y, 'x--', label='Line of sight')
ax.plot(position[0], position[1], 'd', color='C0')
ax.legend()
ax.set_aspect('equal')
ax.set_xlabel('$X$ [m]')
ax.set_ylabel('$Y$ [m]')

# Poloidal cross-section view
ax = axs[1]
ax.plot(eq.first_wall.R, eq.first_wall.Z, 'k-')
ax.plot(eq.lcfs.R, eq.lcfs.Z, 'k--')
ax.plot(LOS.R, LOS.Z, 'x--')
ax.plot(LOS.R[0], position[2], 'd', color='C0')
ax.set_aspect('equal')
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel('$R$ [m]')
ax.set_ylabel('$Z$ [m]')
```

[25]:
```
Text(0, 0.5, '$Z$ [m]')
```



2.2.8 Field line tracing

In this section, we show how to trace field lines and calculate their length. (In the core plasma, the length is defined as the parallel distance of one poloidal turn. In the SOL, it's the so-called connection length.) First we define a set of five starting points, all located at the outer midplane ($Z = 0$) with R going from $0.55\,\mathrm{m}$ (core) to $0.76\,\mathrm{m}$ (SOL).

[26]:
```
# Define the starting points
N = 5
Rs = np.linspace(0.57, 0.76, N, endpoint=True)
Zs = np.zeros_like(Rs)
```

Next, the field lines beginning at these points are traced. The default tracing direction is `direction=1`, that is, following the direction of the toroidal magnetic field.

[27]:
```
traces = eq.trace_field_line(R=Rs, Z=Zs)
```

```
direction: 1
dphidtheta: 1.0
direction: 1
dphidtheta: 1.0
direction: 1
dphidtheta: 1.0
direction: 1
dphidtheta: 1.0
```

To visualise the field lines, we plot them in top view, poloidal cross-section view and 3D view.

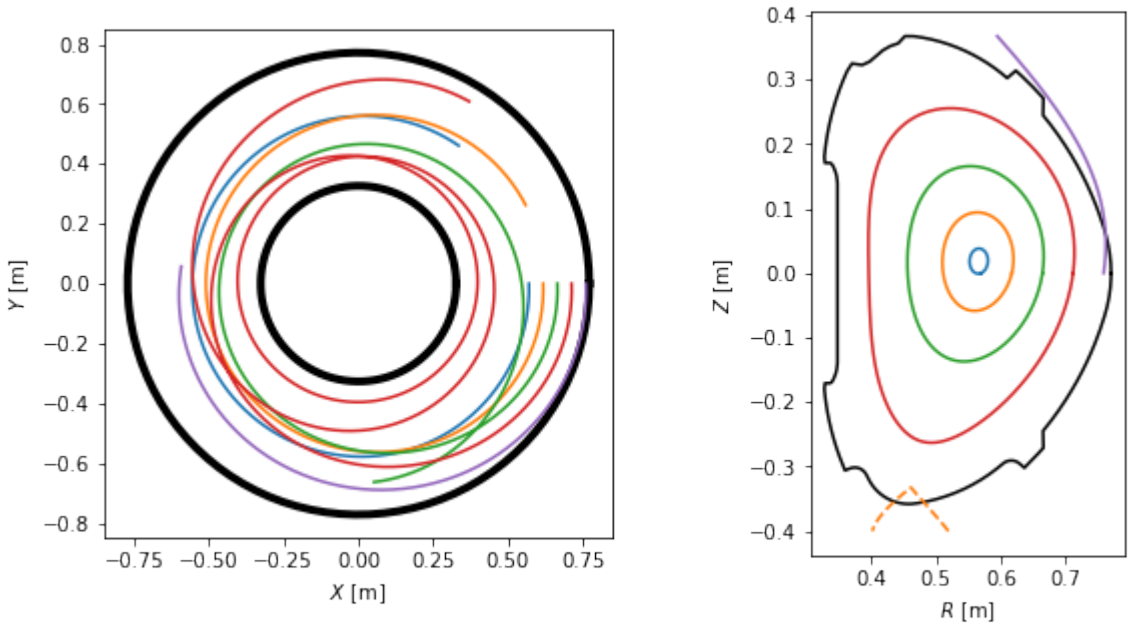
[28]:
```
# Define limiter as viewed from the top
Ns = 100
inner_lim = eq.coordinates(np.min(eq.first_wall.R)*np.ones(Ns), np.zeros(Ns), np.
↪linspace(0, 2*np.pi, Ns))
outer_lim = eq.coordinates(np.max(eq.first_wall.R)*np.ones(Ns), np.zeros(Ns), np.
↪linspace(0, 2*np.pi, Ns))

fig = plt.figure(figsize=(10,5))

#Plot top view of the field lines
```

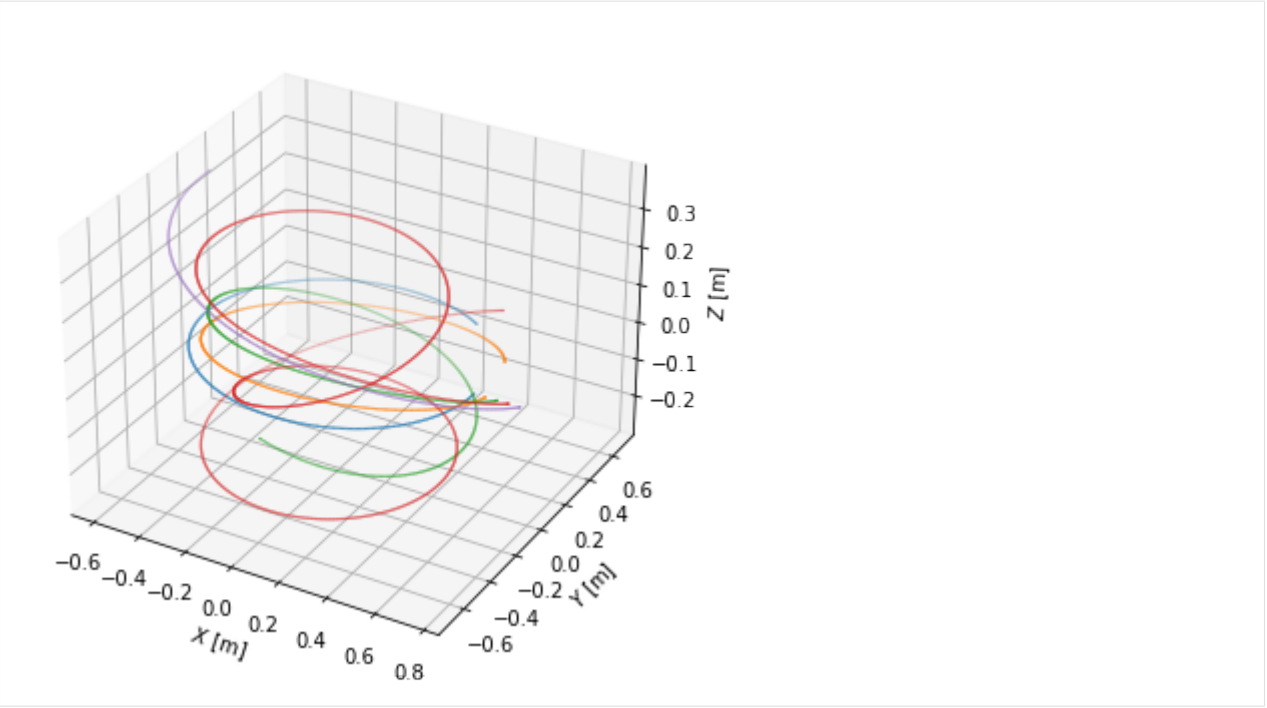
(continues on next page)

(continued from previous page)

```
ax = plt.subplot(121)
plt.plot(inner_lim.X, inner_lim.Y, 'k-', lw=4)
plt.plot(outer_lim.X, outer_lim.Y, 'k-', lw=4)
for fl in traces:
    ax.plot(fl.X, fl.Y)
ax.set_xlabel('$X$ [m]')
ax.set_ylabel('$Y$ [m]')
ax.set_aspect('equal')

#Plot poloidal cross-section view of the field lines
ax = plt.subplot(122)
plt.plot(eq.first_wall.R, eq.first_wall.Z, 'k-')
plt.plot(eq.separatrix.R, eq.separatrix.Z, 'C1--')
for fl in traces:
    plt.plot(fl.R, fl.Z)
ax.set_xlabel('$R$ [m]')
ax.set_ylabel('$Z$ [m]')
ax.set_aspect('equal')

#Plot 3D view of the field lines
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure(figsize=(6,6))
ax = fig.gca(projection='3d')
for fl in traces:
    ax.scatter(fl.X, fl.Y, fl.Z, s=0.3, marker='.')
ax.set_xlabel('$X$ [m]')
ax.set_ylabel('$Y$ [m]')
ax.set_zlabel('$Z$ [m]')
#ax.set_aspect('equal')
```

[28]:
```
Text(0.5, 0, '$Z$ [m]')
```





One may calculate the field line length using the attribute `length`. To demonstrate the connection length profile, we define a couple more SOL field lines. Note that now the `direction` argument changes whether we trace to the HFS or LFS limiter/divertor. Also pay attention to the `in_first_wall=True` argument, which tells the field lines to terminate upon hitting the first wall. (Otherwise they would be terminated at the edge of a rectangle surrounding the vacuum vessel.)

[29]:
```
Rsep = 0.7189 # You might want to change this when switching between different test␣
↪equilibria.
Rs_SOL = Rsep + 0.001*np.array([0, 0.2, 0.5, 0.7, 1, 1.5, 2.5, 4, 6, 9, 15, 20])
Zs_SOL = np.zeros_like(Rs_SOL)

SOL_traces = eq.trace_field_line(R=Rs_SOL, Z=Zs_SOL, direction=-1, in_first_wall=True)
```

Finally we calculate the connection length and plot its profile.

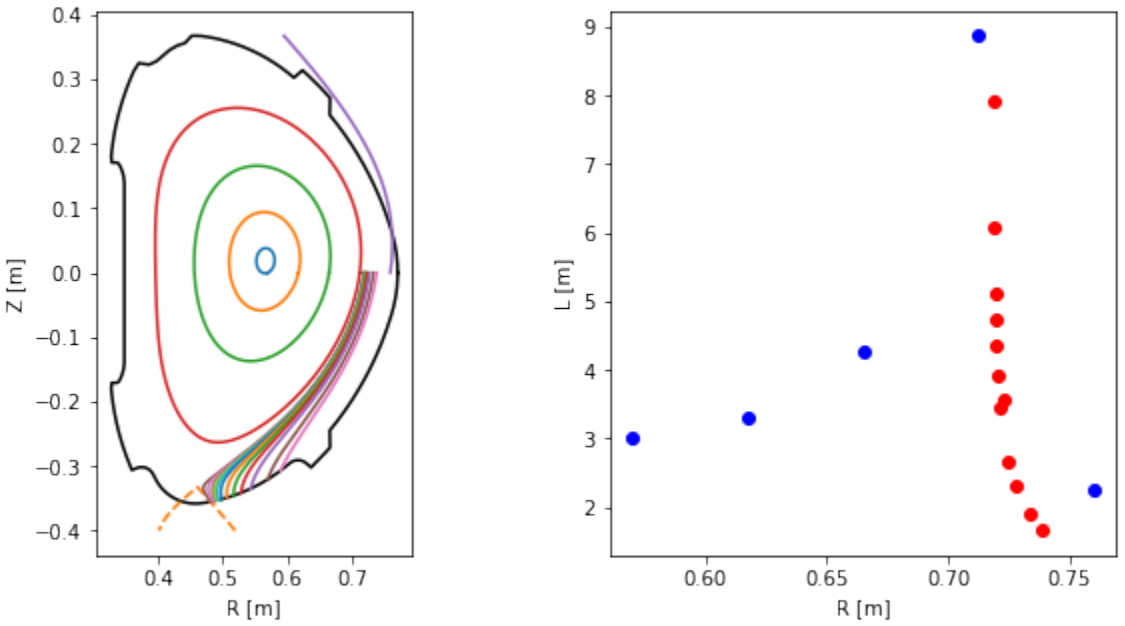
[30]:
```
#Calculate field line length
L = np.array([traces[k].length for k in range(N)])
L_conn = np.array([SOL_traces[k].length for k in range(len(SOL_traces))])

fig = plt.figure(figsize=(10,5))

#Plot poloidal cross-section view of the field lines
ax = plt.subplot(121)
ax.plot(eq.first_wall.R, eq.first_wall.Z, 'k-')
ax.plot(eq.separatrix.R, eq.separatrix.Z, 'C1--')
for fl in np.hstack((traces, SOL_traces)):
    ax.plot(fl.R, fl.Z)
ax.set_xlabel('R [m]')
ax.set_ylabel('Z [m]')
ax.set_aspect('equal')

#Plot connection length profile
```

(continues on next page)

(continued from previous page)

```
ax = plt.subplot(122)
ax.plot(Rs, L, 'bo')
ax.plot(Rs_SOL, L_conn, 'ro')
ax.set_xlabel('R [m]')
ax.set_ylabel('L [m]')
```

[30]:
```
Text(0, 0.5, 'L [m]')
```



2.2.9 Straight field lines

In the field of MHD, it is sometimes advantageous to go from the normal toroidal coordinates $[R, \theta, \phi]$ to a coordinate system $[R, \theta^*, \phi]$ where field lines are straight. In this section, we show how to define such a coordinate system using PLEQUE.

The field line we are going to visualise is on the resonant surface $q = 5/3$ (and therefore it closes upon itself after three poloidal turns). First, we find the Ψ_N of this surface.

[31]:
```
from scipy.optimize import minimize_scalar, brentq

#Find the Psi_N where the safety factor is 5/3
psi_onq = brentq(lambda psi_n: np.abs(eq.q(psi_n)) - 5/3, 0, 0.95)
print(r'Psi_N = {:.3f}'.format(psi_onq))

#Define the resonant flux surface using this Psi_N
surf = eq._flux_surface(psi_n = psi_onq)[0]
```

```
Psi_N = 0.714
```

[32]:
```
from scipy.interpolate import CubicSpline
from numpy import ma #module for masking arrays

#Define the normal poloidal coordinate theta (and subtract 2*pi from any value that␣
↪exceeds 2*pi)
```

(continues on next page)

(continued from previous page)

```
theta = np.mod(surf.theta, 2*np.pi)

#Define the special poloidal coordinate theta_star and
theta_star = surf.straight_fieldline_theta

#Sort the two arrays to start at theta=0 and decrease their spatial resolution by 75 %
asort = np.argsort(theta)
#should be smothed
theta = theta[asort][2::4]
theta_star = theta_star[asort][2::4]

#Interpolate theta_star with a periodic spline
thstar_spl = CubicSpline(theta, theta_star, extrapolate='periodic')
```

Now we trace a field line along the resonant magnetic surface, starting at the midplane (the intersection of the resonant surface with the horizontal plane passing through the magnetic axis). Since the field line is within the confined plasma, the tracing terminates after one poloidal turn. We begin at the last point of the field line and restart the tracing two more times, obtaining a full field line which closes into itself.

[33]:
```
tr1 = eq.trace_field_line(r=eq.coordinates(psi_onq).r_mid[0], theta=0.09)[0]
tr2 = eq.trace_field_line(tr1.R[-1], tr1.Z[-1], tr1.phi[-1])[0]
tr3 = eq.trace_field_line(tr2.R[-1], tr2.Z[-1], tr2.phi[-1])[0]
```

```
direction: 1
dphidtheta: 1.0
direction: 1
dphidtheta: 1.0
direction: 1
dphidtheta: 1.0
```

We visualise the field lines in top view, poloidal cross-section view and 3D view. Notice that the field lines make five toroidal turns until they close in on themselves, which corresponds to the $m = 5$ resonant surface.

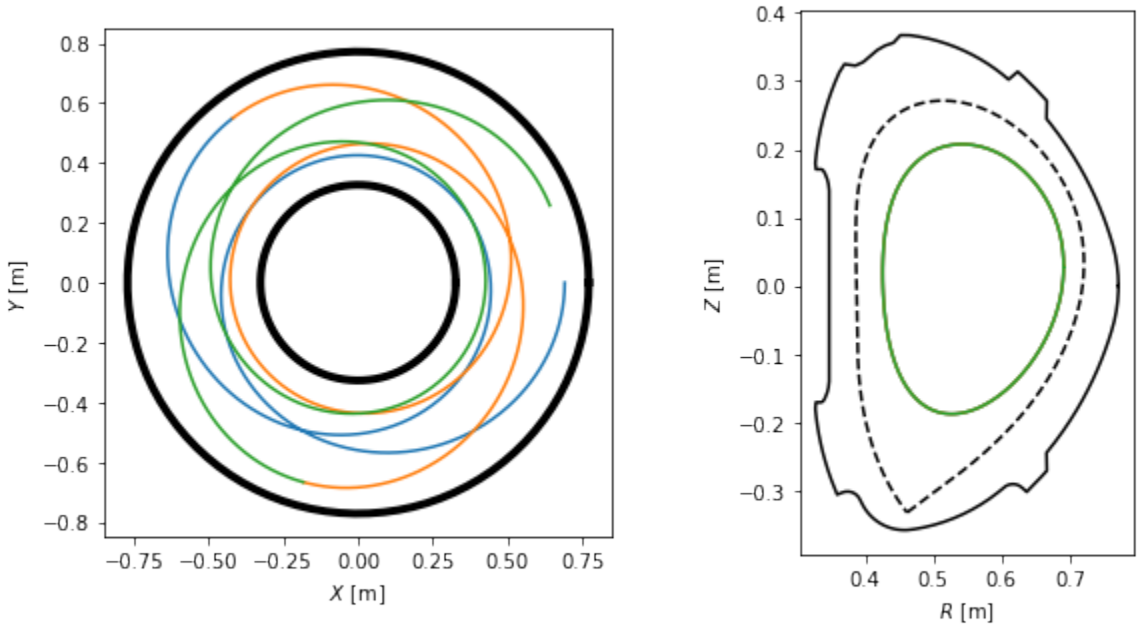
[34]:
```
plt.figure(figsize=(10,5))

# Define limiter as viewed from the top
Ns = 100
inner_lim = eq.coordinates(np.min(eq.first_wall.R)*np.ones(Ns), np.zeros(Ns), np.
↪linspace(0, 2*np.pi, Ns))
outer_lim = eq.coordinates(np.max(eq.first_wall.R)*np.ones(Ns), np.zeros(Ns), np.
↪linspace(0, 2*np.pi, Ns))

#Plot the field lines in top view
ax = plt.subplot(121)
ax.plot(inner_lim.X, inner_lim.Y, 'k-', lw=4)
ax.plot(outer_lim.X, outer_lim.Y, 'k-', lw=4)
ax.plot(tr1.X, tr1.Y)
ax.plot(tr2.X, tr2.Y)
ax.plot(tr3.X, tr3.Y)
ax.set_xlabel('$X$ [m]')
ax.set_ylabel('$Y$ [m]')
ax.set_aspect('equal')

#Plot the field lines in the poloidal cross-section view
ax = plt.subplot(122)
ax.plot(eq.first_wall.R, eq.first_wall.Z, 'k-')
```

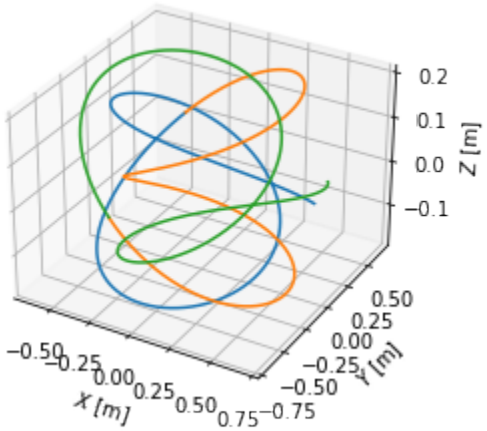
(continues on next page)

(continued from previous page)

```
ax.plot(eq.lcfs.R, eq.lcfs.Z, 'k--')
ax.plot(tr1.R, tr1.Z)
ax.plot(tr2.R, tr2.Z)
ax.plot(tr3.R, tr3.Z)
ax.set_xlabel('$R$ [m]')
ax.set_ylabel('$Z$ [m]')
ax.set_aspect('equal')

#Plot the field line in 3D
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot(tr1.X, tr1.Y, tr1.Z)
ax.plot(tr2.X, tr2.Y, tr2.Z)
ax.plot(tr3.X, tr3.Y, tr3.Z)
#ax.set_aspect('equal')
ax.set_xlabel('$X$ [m]')
ax.set_ylabel('$Y$ [m]')
ax.set_zlabel('$Z$ [m]')
```

[34]:
```
Text(0.5, 0, '$Z$ [m]')
```

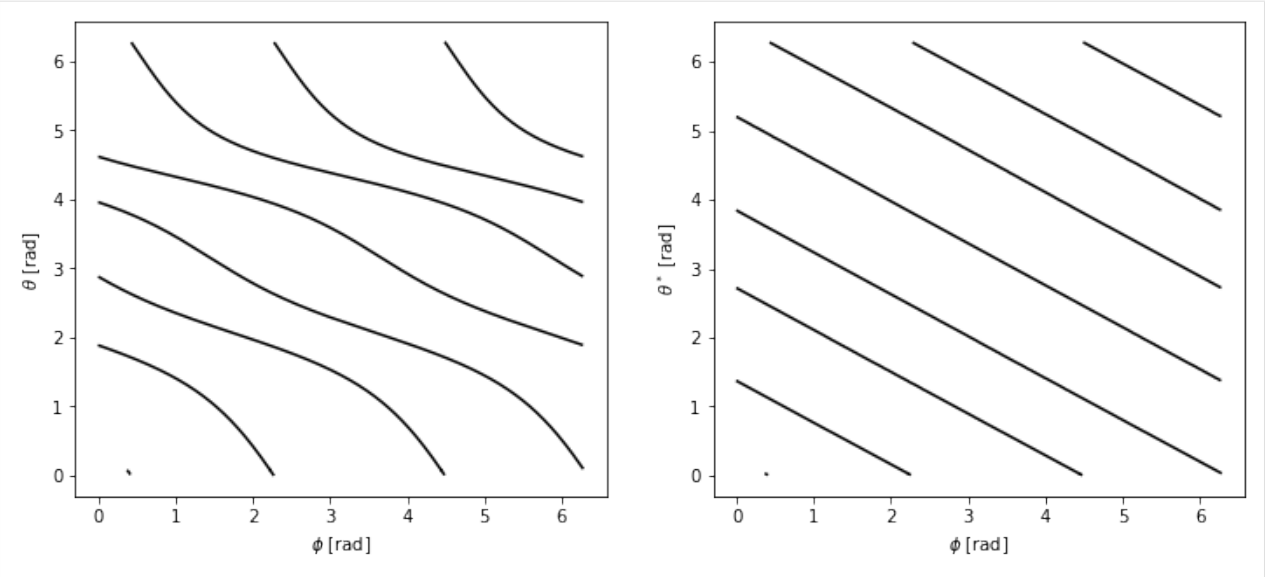




Plotting the field lines in the $[\theta, \phi]$ and $[\theta^*, \phi]$ coordinates, we find that they are curves in the former and straight lines in the latter.

[35]:
```
fig, axes = plt.subplots(1, 2, figsize=(12,5))
ax1, ax2 = axes

for t in [tr1, tr2, tr3]:
    # Extract the theta, theta_star and Phi coordinates from the field lines
    theta = np.mod(t.theta, 2*np.pi)
    theta_star = thstar_spl(theta)
    phi = np.mod(t.phi, 2*np.pi)

    # Mask the coordinates for plotting purposes
    theta = ma.masked_greater(theta, 2*np.pi-1e-2)
    theta = ma.masked_less(theta, 1e-2)
    theta_star = ma.masked_greater(theta_star, 2*np.pi-1e-2)
    theta_star = ma.masked_less(theta_star, 1e-2)
    phi = ma.masked_greater(phi, 2*np.pi-1e-2)
    phi = ma.masked_less(phi, 1e-2)

    # Plot the coordinates [theta, Phi] and [theta_star, Phi]
    ax1.plot(phi, theta, 'k-')
    ax2.plot(phi, theta_star, 'k-')

#Add labels to the two subplots
ax1.set_xlabel(r'$\phi$ [rad]')
ax1.set_ylabel(r'$\theta$ [rad]')
ax2.set_xlabel(r'$\phi$ [rad]')
ax2.set_ylabel(r'$\theta^*$ [rad]')
```

[35]:
```
Text(0, 0.5, '$\\theta^*$ [rad]')
```



Finally, we plot the difference between the two coordinate systems in the poloidal cross-section view, where lines represent points with constant ψ_N and θ (or θ^*).

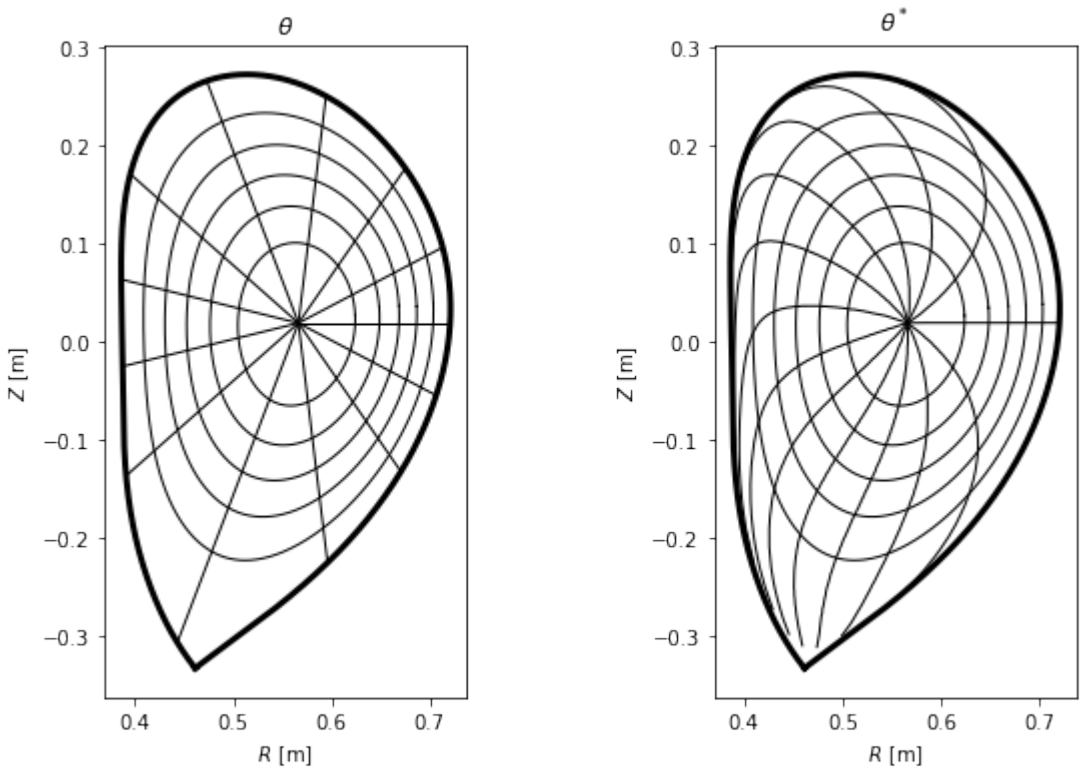
[36]:
```
#Define flux surfaces where theta will be evaluated
psi_n = np.linspace(0, 1, 200)[1:-1]
surfs = [eq._flux_surface(pn)[0] for pn in psi_n]

#Define the flux surfaces which will show on the plot
psi_n2 = np.linspace(0, 1, 7)[1:]
surfs2 = [eq._flux_surface(pn)[0] for pn in psi_n2]

#Define poloidal angles where theta isolines will be plotted
thetas = np.linspace(0, 2*np.pi, 13, endpoint=False)

#Prepare figure
fig, axes = plt.subplots(1, 2, figsize=(10,6))
ax1, ax2 = axes

#Plot LCFS and several flux surfaces in both the plots
eq.lcfs.plot(ax = ax1, color = 'k', ls = '-', lw=3)
eq.lcfs.plot(ax = ax2, color = 'k', ls = '-', lw=3)
for s in surfs2:
    s.plot(ax = ax1, color='k', lw = 1)
    s.plot(ax = ax2, color='k', lw = 1)

#Plot theta and theta_star isolines
for th in thetas:
    # this is so ugly it has to implemented better as soon as possible (!)
#    print(th)
    c = eq.coordinates(r = np.linspace(0, 0.4, 150), theta = np.ones(150)*th)
    amin = np.argmin(np.abs(c.psi_n - 1))
    r_lcfs = c.r[amin]

    psi_n = np.array([np.mean(s.psi_n) for s in surfs])
    c = eq.coordinates(r = np.linspace(0, r_lcfs, len(psi_n)), theta=np.ones(len(psi_
↪n))*th)
    c.plot(ax = ax1, color='k', lw=1)
```

(continues on next page)

(continued from previous page)

```
    idxs = [np.argmin(np.abs(s.straight_fieldline_theta - th)) for s in surfs]
    rs = [s.r[i] for s,i in zip(surfs,idxs)]
    rs = np.hstack((0, rs))
    thetas  = [s.theta[i] for s,i in zip(surfs,idxs)]
    thetas = np.hstack((0, thetas))
    c = eq.coordinates(r = rs, theta = thetas)
    c.plot(ax = ax2, color = 'k', lw=1)

# Make both the subplots pretty
ax1.set_title(r'$\theta$')
ax1.set_aspect('equal')
ax1.set_xlabel('$R$ [m]')
ax1.set_ylabel('$Z$ [m]')
ax2.set_title(r'$\theta^*$')
ax2.set_aspect('equal')
ax2.set_xlabel('$R$ [m]')
ax2.set_ylabel('$Z$ [m]')
```

[36]:
```
Text(0, 0.5, '$Z$ [m]')
```



2.2.10 Separatrix position in a profile

In experiment, one is often interested where the separatrix is along the chord of their measurement. In the following example the separatrix coordinates are calculated at the geometric outer midplane, that is, $Z = 0$.

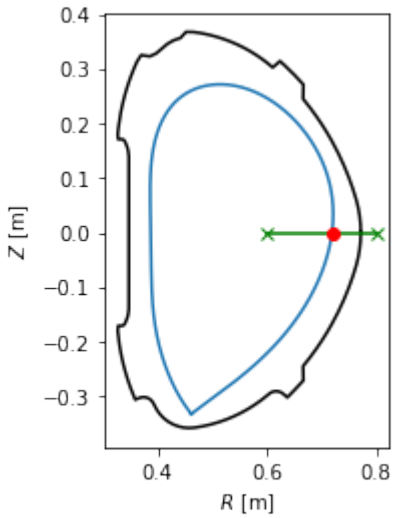
[37]:
```
#Define the measurement chord using two points
chord = eq.coordinates(R=[0.6,0.8], Z=[0,0])
```

(continues on next page)

(continued from previous page)

```
#Calculate the intersection of the chord with the separatrix in 2D
intersection_point = chord.intersection(eq.lcfs, dim=2)

#Plot the plasma with the intersection point
ax = plt.gca()
eq.lcfs.plot()
eq.first_wall.plot(c='k')
chord.plot(color='g', marker='x')
intersection_point.plot(marker='o', color='r')
ax.set_aspect('equal')
ax.set_xlabel('$R$ [m]')
ax.set_ylabel('$Z$ [m]')

intersection_point.R
```

[37]:
```
array([0.71882008])
```



[ ]:

The notebooks are stored in the `notebooks` folder in the source code. More examples can be found in the `examples` folder.

CHAPTER 3

Coordinates

3.1 Accepted coordinates types

1D - coordinates

Coordinate	Code	Note
ψ_{N}	`psi_n`	Default 1D coordinate
ψ	`psi`	
ρ	`rho`	$\rho = \sqrt{\psi_n}$

2D - coordinates

Coordinate	Code	Note
(R, Z)	`R, Z`	Default 2D coordinate
(r, θ)	`r, theta`	Polar coordinates with respect to magnetic axis

3D - coordinates

Coordinate	Code	Note
(R, Z, ϕ)	`R, Z, phi`	Default 3D coordinate
(X, Y, Z)	`X, Y, Z`	

CHAPTER 4

Flux expansion module

PLEQUE provides set of functions for mapping of upstream heat fluxes.

4.1 API Reference

pleque.utils.flux_expansions.**effective_poloidal_heat_flux_exp_coef**(*equilibrium: pleque.core.equilibrium.Equilibriu*
coords: pleque.core.coordinates.Coordinat

Effective poloidal heat flux expansion coefficient

Definition:

$$f_{\mathrm{pol,heat,eff}} = \frac{B_\theta^{\mathrm{u}}}{B_\theta^{\mathrm{t}}} \frac{1}{\sin\beta} = \frac{f_{\mathrm{pol}}}{\sin\beta}$$

Where β is inclination angle of the poloidal magnetic field and the target plane.

Typical usage:

Effective poloidal heat flux expansion coefficient is typically used scale upstream poloidal heat flux to the target plane.

$$q_\perp^{\mathrm{t}} = \frac{q_\theta^{\mathrm{u}}}{f_{\mathrm{pol,heat,eff}}}$$

Parameters

- **equilibrium** – Instance of `Equilibrium`.
- **coords** – `Coordinates` where the coefficient is evaluated.

Returns

pleque.utils.flux_expansions.**effective_poloidal_mag_flux_exp_coef**(*equilibrium: pleque.core.equilibrium.Equilibrium*, *coords: pleque.core.coordinates.Coordinates*)

Effective poloidal magnetic flux expansion coefficient

Definition:

$$f_{\mathrm{pol,eff}} = \frac{B_\theta^{\mathrm{u}} R^{\mathrm{u}}}{B_\theta^{\mathrm{t}} R^{\mathrm{t}}} \frac{1}{\sin\beta} = \frac{f_{\mathrm{pol}}}{\sin\beta}$$

Where β is inclination angle of the poloidal magnetic field and the target plane.

Typical usage:

Effective magnetic flux expansion coefficient is typically used for λ scaling of the target λ with respect to the upstream value.

$$\lambda^{\mathrm{t}} = \lambda_q^{\mathrm{u}} f_{\mathrm{pol,eff}}$$

This coefficient can be also used to calculate peak target heat flux from the total power through LCFS if the perpendicular diffusion is neglected. Then for the peak value stays

$$q_{\perp,\mathrm{peak}} = \frac{P_{\mathrm{div}}}{2\pi R^{\mathrm{t}} \lambda_q^{\mathrm{u}}} \frac{1}{f_{\mathrm{pol,eff}}}$$

Where P_{div} is total power to outer strike point and $lambda_q^mathrm{u}$ is e-folding length on the outer midplane.

Parameters

- **equilibrium** – Instance of `Equilibrium`.
- **coords** – `Coordinates` where the coefficient is evaluated.

Returns

pleque.utils.flux_expansions.**impact_angle_cos_pol_projection**(*coords: pleque.core.coordinates.Coordinates*)

Impact angle calculation - dot product of PFC norm and local magnetic field direction poloidal projection only. Internally uses *incidence_angle_sin* function where *vecs* are replaced by the vector of the poloidal magnetic field (Bphi = 0).

Returns `array` of impact angles

pleque.utils.flux_expansions.**impact_angle_sin**(*coords: pleque.core.coordinates.Coordinates*)

Impact angle calculation - dot product of PFC norm and local magnetic field direction. Internally uses *incidence_angle_sin* function where *vecs* are replaced by the vector of the magnetic field.

Returns `array` of impact angles cosines

pleque.utils.flux_expansions.**incidence_angle_sin**(*coords: pleque.core.coordinates.Coordinates*, *vecs*)

Parameters

- **coords** – `Coordinate` object (of length `N_vecs`)of a line in the space on which the incidence angle is evaluated.
- **vecs** – `array (3, N_vecs)` vectors in (R, Z, phi) space.

Returns array of sines of angles of incidence. I.e. cosine of the angle between the normal to the line (in the poloidal plane) and the corresponding vector.

pleque.utils.flux_expansions.**parallel_heat_flux_exp_coef**(*equilibrium: pleque.core.equilibrium.Equilibrium*, *coords: pleque.core.coordinates.Coordinates*)

Parallel heat flux expansion coefficient

Definition:

$$f_{\parallel} = \frac{B^{\mathrm{u}}}{B^{\mathrm{t}}}$$

Typical usage:

Parallel heat flux expansion coefficient is typically used to scale total upstream heat flux parallel to the magnetic field along the magnetic field lines.

$$q_{\parallel}^{\mathrm{t}} = \frac{q_{\parallel}^{\mathrm{u}}}{f_{\parallel}}$$

Parameters

- **equilibrium** – Instance of `Equilibrium`.
- **coords** – `Coordinates` where the coefficient is evaluated.

Returns

pleque.utils.flux_expansions.**poloidal_heat_flux_exp_coef**(*equilibrium: pleque.core.equilibrium.Equilibrium*, *coords: pleque.core.coordinates.Coordinates*)

Poloidal heat flux expansion coefficient

Definition:

$$f_{\mathrm{pol,heat}} = \frac{B_{\theta}^{\mathrm{u}}}{B_{\theta}^{\mathrm{t}}}$$

Typical usage: *Poloidal heat flux expansion coefficient* is typically used to scale poloidal heat flux (heat flux projected along poloidal magnetic field) along the magnetic field line.

$$q_{\theta}^{\mathrm{t}} = \frac{q_{\theta}^{\mathrm{u}}}{f_{\mathrm{pol,heat}}}$$

Parameters

- **equilibrium** – Instance of `Equilibrium`.
- **coords** – `Coordinates` where the coefficient is evaluated.

Returns

pleque.utils.flux_expansions.**poloidal_mag_flux_exp_coef**(*equilibrium: pleque.core.equilibrium.Equilibrium*, *coords: pleque.core.coordinates.Coordinates*)

Poloidal magnetic flux expansion coefficient.

Definition:

$$f_{\mathrm{pol}} = \frac{\Delta r^{\mathrm{t}}}{\Delta r^{\mathrm{u}}} = \frac{B_{\theta}^{\mathrm{u}} R^{\mathrm{u}}}{B_{\theta}^{\mathrm{t}} R^{\mathrm{t}}}$$

Typical usage:

Poloidal magnetic flux expansion coefficient is typically used for λ scaling in plane perpendicular to the poloidal component of the magnetic field.

Parameters

- **equilibrium** – Instance of `Equilibrium`.
- **coords** – `Coordinates` where the coefficient is evaluated.

Returns

pleque.utils.flux_expansions.**total_heat_flux_exp_coef**(*equilibrium: pleque.core.equilibrium.Equilibrium*, *coords: pleque.core.coordinates.Coordinates*)

Total heat flux expansion coefficient

Definition:

$$f_{\mathrm{tot}} = \frac{B^{\mathrm{u}}}{B^{\mathrm{t}}} \frac{1}{\sin\alpha} = \frac{f_{\parallel}}{\sin\alpha}$$

Where α is an inclination angle of the total magnetic field and the target plane.

Important: α is an inclination angle of the total magnetic field to the target plate. Whereas β is an inclination of poloidal components of the magnetic field to the target plate.

Typical usage:

Total heat flux expansion coefficient is typically used to project total upstream heat flux parallel to the magnetic field to the target plane.

$$q_{\perp}^{\mathrm{t}} = \frac{q_{\parallel}^{\mathrm{u}}}{f_{\mathrm{tot}}}$$

Parameters

- **equilibrium** – Instance of `Equilibrium`.
- **coords** – `Coordinates` where the coefficient is evaluated.

Returns

4.2 References

Theiler, C., et al.: *Results from recent detachment experiments in alternativee divertor cofigurations on TCV*, Nucl. Fusion **57** (2017) 072008 16pp

Vondracek, P.: *Plasma Heat Flux to Solid Structures in Tokamaks*, PhD thesis, Prague 2019

CHAPTER 5

Naming convention used in PLEQUE

5.1 Coordinates

Here presented naming convention is used to read/create input/output dict/xarray files.

- **2D**
  - `R` (default): Radial cylindrical coordinates with zero on machine axis
  - `Z` (default): Vertical coordinate with zero on machine geometrical axis
- **1D**
  - `psi_n` (default): Normalized poloidal magnetic flux with zero on magnetic axis and one on the last closed flux surface

    $\psi_{\mathrm{N}} = \frac{\psi - \psi_{\mathrm{ax}}}{\psi_{\mathrm{LCFS}} - \psi_{\mathrm{ax}}}$
  - Fallowing input options are not implemented yet.
  - `rho`: $\rho = \sqrt{\psi_{\mathrm{N}}}$
  - `psi_1dprof` - poloidal magnetic flux; this coordinate axis is used only if `psi_n` is not found on the input. Output files uses implicitly `psi_n` axis.

5.2 2D profiles

- **Required on the input**
  - `psi` (Wb): poloidal magnetic flux
- **Calculated**
  - `B_R` (T): R component of the magnetic field.
  - `B_Z` (T): Z component of the magnetic field.

  - `B_pol` (T): Poloidal component of the magnetic field. $B_\theta = \text{sign}\,(I_p)\sqrt{B_R^2 + B_Z^2}$ **Todo** resolve the sign of B_pol and implement it!!!
  - `B_tor` (T): Toroidal component of the magnetic field.
  - `B_abs` (T): Absolute value of the magnetic field.
  - `j_R` (A/m2): R component of the current density. **todo: Check the current unit**
  - `j_Z` (A/m2): Z component of the current density.
  - `j_pol` (A/m2): Poloidal component of the current density.
  - `j_tor` (A/m2): Toroidal component of the current density.
  - `j_abs` (A/m2): Asolute value of the current density.

5.3 1D profiles

- **Required on the input**
  - `pressure` (Pa)
  - `pprime` (Pa/Wb)
  - `F`: $F = RB_\phi$
- **Calculated**
  - `pprime`: $p\partial_\psi$
  - `Fprime`: $F' = \partial_\psi F$
  - `FFprime`: $FF' = F\partial_\psi F$
  - `fprime`: $f' = \partial_\psi f$
  - `f`: $f = (1/\mu_0)RB_\phi$
  - `ffprime`: $ff' = f\partial_\psi f$
  - `rho`, `psi_n`
- **Deriver**
  - q: safety factor profile
  - *qprime*$q' = \partial_\psi q$
  - **Not yet implemented:**
    - *magnetic_shear*
    - ...

5.4 Attributes

- To be written.

5.5 FluxSurface quantities

CHAPTER 6

API Reference

6.1 API Reference

6.1.1 Equilibrium

class pleque.core.equilibrium.**Equilibrium**(*basedata: xarray.core.dataset.Dataset*, *first_wall=None*, *mg_axis=None*, *psi_lcfs=None*, *x_points=None*, *strike_points=None*, *init_method='hints'*, *spline_order=3*, *spline_smooth=0*, *cocos=3*, *verbose=False*)

Bases: object

Equilibrium class ...

B_R(**coordinates*, *R=None*, *Z=None*, *coord_type=('R', 'Z')*, *grid=True*, ***coords*)

Poloidal value of magnetic field in Tesla.

Parameters

- **coordinates** –
- **R** –
- **Z** –
- **coord_type** –
- **grid** –
- **coords** –

Returns

B_Z(**coordinates*, *R=None*, *Z=None*, *coord_type=None*, *grid=True*, ***coords*)

Poloidal value of magnetic field in Tesla.

Parameters

- **grid** –
- **coordinates** –
- **R** –
- **Z** –
- **coord_type** –
- **coords** –

Returns

B_abs (**coordinates*, *R=None*, *Z=None*, *coord_type=None*, *grid=True*, ***coords*)
Absolute value of magnetic field in Tesla.

Parameters

- **grid** –
- **coordinates** –
- **R** –
- **Z** –
- **coord_type** –
- **coords** –

Returns Absolute value of magnetic field in Tesla.

B_pol (**coordinates*, *R=None*, *Z=None*, *coord_type=None*, *grid=True*, ***coords*)
Absolute value of magnetic field in Tesla.

Parameters

- **grid** –
- **coordinates** –
- **R** –
- **Z** –
- **coord_type** –
- **coords** –

Returns

B_tor (**coordinates*, *R: numpy.array = None*, *Z: numpy.array = None*, *coord_type=None*, *grid=True*, ***coords*)
Toroidal value of magnetic field in Tesla.

Parameters

- **grid** –
- **coordinates** –
- **R** –
- **Z** –
- **coord_type** –
- **coords** –

Returns

Bvec(**coordinates*, *swap_order=False*, *R=None*, *Z=None*, *coord_type=None*, *grid=True*, ***coords*)
Magnetic field vector

Parameters

- **grid** –
- **coordinates** –
- **swap_order** – bool,
- **R** –
- **Z** –
- **coord_type** –
- **coords** –

Returns Magnetic field vector array (3, N) if swap_order is False.

Bvec_norm(**coordinates*, *swap_order=False*, *R=None*, *Z=None*, *coord_type=None*, *grid=True*, ***coords*)
Magnetic field vector, normalised

Parameters

- **grid** –
- **coordinates** –
- **swap_order** –
- **R** –
- **Z** –
- **coord_type** –
- **coords** –

Returns Normalised magnetic field vector array (3, N) if swap_order is False.

F(**coordinates*, *R=None*, *Z=None*, *psi_n=None*, *coord_type=None*, *grid=True*, ***coords*)

FFprime(**coordinates*, *R=None*, *Z=None*, *psi_n=None*, *coord_type=None*, *grid=True*, ***coords*)

Fprime(**coordinates*, *R=None*, *Z=None*, *psi_n=None*, *coord_type=None*, *grid=True*, ***coords*)

Parameters

- **coordinates** –
- **R** –
- **Z** –
- **psi_n** –
- **coord_type** –
- **grid** –
- **coords** –

Returns

I_plasma
Toroidal plasma current. Calculated as toroidal current through the LCFS.

Returns (float) Value of toroidal plasma current.

__init__(*basedata: xarray.core.dataset.Dataset*, *first_wall=None*, *mg_axis=None*, *psi_lcfs=None*, *x_points=None*, *strike_points=None*, *init_method='hints'*, *spline_order=3*, *spline_smooth=0*, *cocos=3*, *verbose=False*)

Equilibrium class instance should be obtained generally by functions in pleque.io package.

Optional arguments may help the initialization.

Parameters

- **basedata** – xarray.Dataset with psi(R, Z) on a rectangular R, Z grid, f(psi_norm), p(psi_norm) f = B_tor * R
- **first_wall** – array-like (Nwall, 2) required for initialization in case of limiter configuration.
- **mg_axis** – suspected position of the o-point
- **psi_lcfs** –
- **x_points** –
- **strike_points** –
- **init_method** – str On of ("full", "hints", "fast_forward"). If "full" no hints are taken and module tries to recognize all critical points itself. If "hints" module use given optional arguments as a help with initialization. If "fast-forward" module use given optional arguments as final and doesn't try to correct. *Note:* Only "hints" method is currently tested.
- **spline_order** –
- **spline_smooth** –
- **cocos** – At the moment module assume cocos to be 3 (no other option). The implemetnation is not fully working. Be aware of signs in the module!
- **verbose** –

abs_q(**coordinates*, *R: numpy.array = None*, *Z: numpy.array = None*, *coord_type=None*, *grid=False*, ***coords*)

Absolute value of q.

Parameters

- **coordinates** –
- **R** –
- **Z** –
- **coord_type** –
- **grid** –
- **coords** –

Returns

cocos

Number of internal COCOS representation.

Returns int

connection_length(**coordinates*, *R: numpy.array = None*, *Z: numpy.array = None*, *coord_type=None*, *direction=1*, ***coords*)

Calculate connection length from given coordinates to first wall

Todo: The field line is traced to min/max value of z of first wall, distance is calculated to the last point before first wall.

Parameters

- **coordinates** –
- **R** –
- **Z** –
- **coord_type** –
- **direction** – if positive trace field line in/cons the direction of magnetic field.
- **stopper** – (None, 'poloidal', 'z-stopper) force to use stopper. If None stopper is automatically chosen based on psi_n coordinate.
- **coords** –

Returns

contact_point
Returns contact point as instance of coordinates for circular plasmas. Returns None otherwise. :return:

coordinates(**coordinates*, *coord_type=None*, *grid=False*, ***coords*)
Return instance of Coordinates. If instances of coordinates is already on the input, just pass it through.

Parameters

- **coordinates** –
- **coord_type** –
- **grid** –
- **coords** –

Returns

diff_psi(**coordinates*, *R=None*, *Z=None*, *psi_n=None*, *coord_type=None*, *grid=False*, ***coords*)
Return the value of $\nabla\psi$. It is positive/negative if the ψ is increasing/decreasing.

Parameters

- **coordinates** –
- **R** –
- **Z** –
- **psi_n** –
- **coord_type** –
- **grid** –
- **coords** –

Returns

diff_q(**coordinates*, *R=None*, *Z=None*, *psi_n=None*, *coord_type=None*, *grid=False*, ***coords*)

Parameters

- **self** –
- **coordinates** –

- **R** –
- **Z** –
- **psi_n** –
- **coord_type** –
- **grid** –
- **coords** –

Returns Derivative of q with respect to psi.

effective_poloidal_heat_flux_exp_coef(**coordinates*, *R=None*, *Z=None*, *coord_type=None*, *grid=True*, ***coords*)

Effective poloidal heat flux expansion coefficient

Definition:

$$f_{\mathrm{pol,heat,eff}} = \frac{B_\theta^{\mathrm{u}}}{B_\theta^{\mathrm{t}}} \frac{1}{\sin\beta} = \frac{f_{\mathrm{pol}}}{\sin\beta}$$

Where β is inclination angle of the poloidal magnetic field and the target plane.

Typical usage:

Effective poloidal heat flux expansion coefficient is typically used scale upstream poloidal heat flux to the target plane.

$$q_\perp^{\mathrm{t}} = \frac{q_\theta^{\mathrm{u}}}{f_{\mathrm{pol,heat,eff}}}$$

Parameters

- **coordinates** –
- **R** –
- **Z** –
- **coord_type** –
- **grid** –
- **coords** –

Returns

effective_poloidal_mag_flux_exp_coef(**coordinates*, *R=None*, *Z=None*, *coord_type=None*, *grid=True*, ***coords*)

Effective poloidal magnetic flux expansion coefficient

Definition:

$$f_{\mathrm{pol,eff}} = \frac{B_\theta^{\mathrm{u}} R^{\mathrm{u}}}{B_\theta^{\mathrm{t}} R^{\mathrm{t}}} \frac{1}{\sin\beta} = \frac{f_{\mathrm{pol}}}{\sin\beta}$$

Where β is inclination angle of the poloidal magnetic field and the target plane.

Typical usage:

Effective magnetic flux expansion coefficient is typically used for λ scaling of the target λ with respect to the upstream value.

$$\lambda^{\mathrm{t}} = \lambda^{\mathrm{u}} f_{\mathrm{pol,eff}}$$

Parameters

- **coordinates** –
- **R** –
- **Z** –
- **coord_type** –
- **grid** –
- **coords** –

Returns

f(**coordinates*, *R=None*, *Z=None*, *psi_n=None*, *coord_type=None*, *grid=True*, ***coords*)

ffprime(**coordinates*, *R=None*, *Z=None*, *psi_n=None*, *coord_type=None*, *grid=True*, ***coords*)

first_wall
If the first wall polygon is composed of 3 and more points Surface instance is returned. If the wall contour is composed of less than 3 points, coordinate instance is returned, because Surface can't be constructed :return:

flux_surface(**coordinates*, *resolution=(0.001, 0.001)*, *dim='step'*, *closed=True*, *inlcfs=True*, *R=None*, *Z=None*, *psi_n=None*, *coord_type=None*, ***coords*)

fluxfuncs

get_precise_lcfs()
Calculate plasma LCFS by field line tracing technique and save LCFS as instance property.

Returns

grid(*resolution=None*, *dim='step'*)
Function which returns 2d grid with requested step/dimensions generated over the reconstruction space.

Parameters

- **resolution** – Iterable of size 2 or a number. If a number is passed, R and Z dimensions will have the same size or step (depending on dim parameter). Different R and Z resolutions or dimension sizes can be required by passing an iterable of size 2. If None, default grid of size (1000, 2000) is returned.
- **dim** – iterable of size 2 or string ('step', 'size'). Default is "step", determines the meaning of the resolution. If "step" used, values in resolution are interpreted as step length in psi poloidal map. If "size" is used, values in resolution are interpreted as requested number of points in a dimension. If string is passed, same value is used for R and Z dimension. Different interpretation of resolution for R, Z dimensions can be achieved by passing an iterable of shape 2.

Returns Instance of *Coordinates* class with grid data

in_first_wall(**coordinates*, *R: numpy.array = None*, *Z: numpy.array = None*, *coord_type=None*, *grid=True*, ***coords*)

in_lcfs(**coordinates*, *R: numpy.array = None*, *Z: numpy.array = None*, *coord_type=None*, *grid=True*, ***coords*)

is_limter_plasma
Return true if the plasma is limited by point or some limiter point.

Returns bool

is_xpoint_plasma
Return true for x-point plasma.

Returns bool

j_R(**coordinates*, *R: numpy.array = None*, *Z: numpy.array = None*, *coord_type=None*, *grid=True*, ***coords*)

j_Z(**coordinates*, *R: numpy.array = None*, *Z: numpy.array = None*, *coord_type=None*, *grid=True*, ***coords*)

j_pol(**coordinates*, *R: numpy.array = None*, *Z: numpy.array = None*, *coord_type=None*, *grid=False*, ***coords*)

Poloidal component of the current density. Calculated as

$$\frac{f'\nabla\psi}{R\mu_0}$$

[Wesson: Tokamaks, p. 105]

Parameters

- **coordinates** –
- **R** –
- **Z** –
- **coord_type** –
- **grid** –
- **coords** –

Returns

j_tor(**coordinates*, *R: numpy.array = None*, *Z: numpy.array = None*, *coord_type=None*, *grid=True*, ***coords*)

todo: to be tested

Toroidal component of the current denisity. Calculated as

$$Rp' + \frac{1}{\mu_0 R} f f'$$

Parameters

- **coordinates** –
- **R** –
- **Z** –
- **coord_type** –
- **grid** –
- **coords** –

Returns

lcfs

limiter_point

The point which "limits" the LCFS of plasma. I.e. contact point in case of limiter plasma and x-point in case of x-point plasma.

Returns Coordinates

magnetic_axis

outter_parallel_fl_expansion_coef(**coordinates*, *R=None*, *Z=None*, *coord_type=None*, *grid=True*, ***coords*)

WIP:Calculate parallel expansion coefitient of the given coordinates with respect to positon on the outer midplane.

outter_poloidal_fl_expansion_coef(**coordinates*, *R=None*, *Z=None*, *coord_type=None*, *grid=True*, ***coords*)

WIP:Calculate parallel expansion coefitient of the given coordinates with respect to positon on the outer midplane.

parallel_heat_flux_exp_coef(**coordinates*, *R=None*, *Z=None*, *coord_type=None*, *grid=True*, ***coords*)

Parallel heat flux expansion coefficient

Definition:

$$f_{\parallel} = \frac{B^{\mathrm{u}}}{B^{\mathrm{t}}}$$

Typical usage:

Parallel heat flux expansion coefficient is typically used to scale total upstream heat flux parallel to the magnetic field along the magnetic field lines.

$$q_{\parallel}^{\mathrm{t}} = \frac{q_{\parallel}^{\mathrm{u}}}{f_{\parallel}}$$

Parameters

- **coordinates** –
- **R** –
- **Z** –
- **coord_type** –
- **grid** –
- **coords** –

Returns

plot_geometry(*axs=None*, ***kwargs*)

Plots the the directions of angles, current and magnetic field.

Parameters

- **axs** – None or tuple of axes. If None new figure with to axes is created.
- **kwargs** – parameters passed to the *plot* routine.

Returns tuple of axis (ax1, ax2)

plot_overview(*ax=None*, ***kwargs*)

Simple routine for plot of plasma overview :return:

poloidal_heat_flux_exp_coef(**coordinates*, *R=None*, *Z=None*, *coord_type=None*, *grid=True*, ***coords*)

Poloidal heat flux expansion coefficient

Definition:

$$f_{\mathrm{pol,heat}} = \frac{B_{\theta}^{\mathrm{u}}}{B_{\theta}^{\mathrm{t}}}$$

Typical usage: *Poloidal heat flux expansion coefficient* is typically used to scale poloidal heat flux (heat flux projected along poloidal magnetic field) along the magnetic field line.

$$q_\theta^{\mathrm{t}} = \frac{q_\theta^{\mathrm{u}}}{f_{\mathrm{pol,heat}}}$$

Parameters

- **coordinates** –
- **R** –
- **Z** –
- **coord_type** –
- **grid** –
- **coords** –

Returns

poloidal_mag_flux_exp_coef(**coordinates*, *R=None*, *Z=None*, *coord_type=None*, *grid=True*, ***coords*)

Poloidal magnetic flux expansion coefficient.

Definition:

$$f_{\mathrm{pol}} = \frac{\Delta r^{\mathrm{t}}}{\Delta r^{\mathrm{u}}} = \frac{B_\theta^{\mathrm{u}} R^{\mathrm{u}}}{B_\theta^{\mathrm{t}} R^{\mathrm{t}}}$$

Typical usage:

Poloidal magnetic flux expansion coefficient is typically used for λ scaling in plane perpendicular to the poloidal component of the magnetic field.

Parameters

- **coordinates** –
- **R** –
- **Z** –
- **coord_type** –
- **grid** –
- **coords** –

Returns

pprime(**coordinates*, *R=None*, *Z=None*, *psi_n=None*, *coord_type=None*, *grid=True*, ***coords*)

pressure(**coordinates*, *R=None*, *Z=None*, *psi_n=None*, *coord_type=None*, *grid=True*, ***coords*)

psi(**coordinates*, *R=None*, *Z=None*, *psi_n=None*, *coord_type=None*, *grid=True*, ***coords*)
Psi value

Parameters

- **psi_n** –
- **coordinates** –
- **R** –
- **Z** –

- **coord_type** –
- **grid** –
- **coords** –

Returns

psi_n(**coordinates*, *R=None*, *Z=None*, *psi=None*, *coord_type=None*, *grid=True*, ***coords*)

q(**coordinates*, *R: numpy.array = None*, *Z: numpy.array = None*, *coord_type=None*, *grid=False*, ***coords*)

r_mid(**coordinates*, *R=None*, *Z=None*, *psi_n=None*, *coord_type=None*, *grid=True*, ***coords*)

rho(**coordinates*, *R=None*, *Z=None*, *psi_n=None*, *coord_type=None*, *grid=True*, ***coords*)

separatrix
If the equilibrium is limited, returns lcfs. If it is diverted it returns separatrix flux surface

Returns

shear(**coordinates*, *R=None*, *Z=None*, *psi_n=None*, *coord_type=None*, *grid=False*, ***coords*)
Normalized magnetic shear parameter

$$\hat{s} =$$

rac{r_mathrm{mid}}{q} rac{mathrm{d}q}{mathrm{d}r}

where r_mathrm{mid} is plasma radius on midplane.

strike_points
Returns contact point if the equilibrium is limited. If the equilibrium is diverted it returns strike points. :return:

surfacefuncs

to_geqdsk(*file*, *nx=64*, *ny=128*, *q_positive=True*, *use_basedata=False*)
Write a GEQDSK equilibrium file.

Parameters

- **file** – str, file name
- **nx** – int
- **ny** – int

tor_flux(**coordinates*, *R: numpy.array = None*, *Z: numpy.array = None*, *coord_type=None*, *grid=False*, ***coords*)
Calculate toroidal magnetic flux Φ from:

$$q =$$

rac{mathrm{d Phi} }{mathrm{d psi}}

param coordinates

param R

param Z

param coord_type

param grid

param coords

return

total_heat_flux_exp_coef(**coordinates*, *R=None*, *Z=None*, *coord_type=None*, *grid=True*, ***coords*)

Total heat flux expansion coefficient

Definition:

$$f_{\mathrm{tot}} = \frac{B^{\mathrm{u}}}{B^{\mathrm{t}}} \frac{1}{\sin\alpha} = \frac{f_{\parallel}}{\sin\alpha}$$

Where α is inclination angle of the total magnetic field and the target plane.

Typical usage:

Total heat flux expansion coefficient is typically used to project total upstream heat flux parallel to the magnetic field to the target plane.

$$q_{\perp}^{\mathrm{t}} = \frac{q_{\parallel}^{\mathrm{u}}}{f_{\mathrm{tot}}}$$

Parameters

- **coordinates** –
- **R** –
- **Z** –
- **coord_type** –
- **grid** –
- **coords** –

Returns

trace_field_line(**coordinates*, *R: numpy.array = None*, *Z: numpy.array = None*, *coord_type=None*, *direction=1*, *stopper_method=None*, *in_first_wall=False*, ***coords*)

Return traced field lines starting from the given set of at least 2d coordinates. One poloidal turn is calculated for field lines inside the separatrix. Outter field lines are limited by z planes given be outermost z coordinates of the first wall.

Parameters

- **coordinates** –
- **R** –
- **Z** –
- **coord_type** –
- **direction** – if positive trace field line in/cons the direction of magnetic field.
- **stopper_method** – (None, 'poloidal', 'z-stopper) force to use stopper. If None stopper is automatically chosen based on psi_n coordinate.
- **in_first_wall** – if True the only inner part of field line is returned.
- **coords** –

Returns

trace_flux_surface(**coordinates*, *s_resolution=0.001*, *R=None*, *Z=None*, *psi_n=None*, *coord_type=None*, ***coords*)

Find a closed flux surface inside LCFS with requested values of psi or psi-normalized.

TODO support open and/or flux surfaces outise LCFS, needs different stopper

Parameters

- **R** –
- **Z** –
- **psi_n** –
- **coord_type** –
- **coordinates** – specifies flux surface to search for (by spatial point or values of psi or psi normalised). If coordinates is spatial point (dim=2) then the trace starts at the midplane. Coordinates.grid must be False.
- **s_resolution** – max_step in the distance along the flux surface contour

Returns FluxSurface

x_point

Return x-point closest in psi to mg-axis if presented on grid. None otherwise.

:return Coordinates

6.1.2 Fluxsurface

class pleque.core.fluxsurface.**FluxSurface**(*equilibrium*, **coordinates*, *coord_type=None*, *grid=False*, ***coords*)

Bases: *pleque.core.fluxsurface.Surface*

__init__(*equilibrium*, **coordinates*, *coord_type=None*, *grid=False*, ***coords*)

Calculates geometrical properties of the flux surface. To make the contour closed, the first and last points in the passed coordinates have to be the same. Instance is obtained by calling method *flux_surface* in instance of *Equilibrium*.

Parameters **coords** – Instance of coordinate class

contains(*coords: pleque.core.coordinates.Coordinates*)

contour

Depracated. Fluxsurface contour points. :return: numpy ndarray

cumsum_surface_average(*func*, *roll=0*)

Return the surface average (over single magnetic surface) value of *func*. Return the value of integration

$$< func > (\psi)_i = \oint_0^{\theta_i} \frac{\mathrm{d}lR}{|\nabla\psi|} a(R, Z)$$

Parameters **func** – func(X, Y), Union[ndarray, int, float]

Returns ndarray

distance(*coords: pleque.core.coordinates.Coordinates*)

elongation

Elongation :return:

eval_q

geom_radius
Geometrical radius a= (R_min + R_max)./2 :return:

get_eval_q(*method*)
Evaluete q usiong formula (5.35) from [Jardin, 2010: Computational methods in Plasma Physics]

Parameters **method** – str, [‘sum’, ‘trapz’, ‘simps’]

Returns

max_radius
maximum radius on the given flux surface :return:

min_radius
minimum radius on the given flux surface :return:

minor_radius
a= (R_min - R_max)./2 :return:

straight_fieldline_theta
Calculate straight field line θ^* coordinate.

Returns

surface_average(*func*, *method='sum'*)
Return the surface average (over single magnetic surface) value of *func*. Return the value of integration

$$< func > (\psi) = \oint \frac{\mathrm{d}lR}{|\nabla\psi|} a(R, Z)$$

Parameters

- **func** – func(X, Y), Union[ndarray, int, float]
- **method** – str, [‘sum’, ‘trapz’, ‘simps’]

Returns

tor_current
Return toroidal current through the closed flux surface

Returns

triangul_low
Lower triangularity :return:

triangul_up
Upper triangularity :return:

triangularity

Returns

class pleque.core.fluxsurface.**Surface**(*equilibrium*, **coordinates*, *coord_type=None*, *grid=False*, ***coords*)
Bases: *pleque.core.coordinates.Coordinates*

__init__(*equilibrium*, **coordinates*, *coord_type=None*, *grid=False*, ***coords*)
Calculates geometrical properties of a specified surface. To make the contour closed, the first and last points in the passed coordinates have to be the same. Instance is obtained by calling method *surface* in instance of *Equilibrium*.

Parameters **coords** – Instance of coordinate class

area
Area of the closed fluxsurface.

Returns

centroid

closed
True if the fluxsurface is closed.

Returns

diff_volume
Diferential volume $V' = dV/d\psi$ Jardin, S.: Computational Methods in Plasma Physics

Returns

length
Length of the fluxsurface contour

Returns

surface
Surface of fluxsurface calculated from the contour length using Pappus centroid theorem : https://en.wikipedia.org/wiki/Pappus%27s_centroid_theorem

Returns float

volume
Volume of the closed fluxsurface calculated from the area using Pappus centroid theorem : https://en.wikipedia.org/wiki/Pappus%27s_centroid_theorem

Returns float

6.1.3 Coordinates

class pleque.core.coordinates.**Coordinates**(*equilibrium*, **coordinates*, *coord_type=None*, *grid=False*, *cocos=None*, ***coords*)

Bases: object

R

R_mid
Major radius on the outer (magnetic) midplane. Major radius is distance from the tokamak axis.

Returns Major radius mapped on the outer midplane.

X

Y

Z

__init__(*equilibrium*, **coordinates*, *coord_type=None*, *grid=False*, *cocos=None*, ***coords*)
Basic PLEQUE class to handle various coordinate systems in tokamak equilibrium.

Parameters

- **equilibrium** –
- ***coordinates** –
  - Can be skipped.
  - array (N, dim) - N points will be generated.
  - One, two are three comma separated one dimensional arrays.
- **coord_type** –

- **grid** –
- **cocos** – Define coordinate system cocos. Id *None* equilibrium default cocos is used. If *equilibrium is None* cocos = 3 (both systems cnt-clockwise) is used.
- ****coords** – Lorem ipsum.

- **1D**: ψ_{N},
- **2D**: (R, Z),
- **3D**: (R, Z, ϕ).

1D - coordinates

Coordinate	Code	Note
ψ_{N}	`psi_n`	Default 1D coordinate
ψ	`psi`	
ρ	`rho`	$\rho = \sqrt{\psi_n}$

2D - coordintares

Coordinate	Code	Note
(R, Z)	`R, Z`	Default 2D coordinate
(r, θ)	`r, theta`	Polar coordinates with respect to magnetic axis

3D - coordinates

Coordinate	Code	Note
(R, Z, ϕ)	`R, Z, phi`	Default 3D coordinate
(X, Y, Z)	`(X, Y, Z)`	Polar coordinates with respect to magnetic axis

as_RZ_mid()
Transforms 1D coordinates to 2D coordinates on the midplane

uses r_mid and the magnetic axis equilibrium

as_array(*dim=None*, *coord_type=None*)
Return array of size (N, dim), where N is number of points and dim number of dimensions specified by coord_type

Parameters

- **dim** – reduce the number of dimensions to dim (todo)
- **coord_type** – not effected at the moment (TODO)

Returns

cum_length
Cumulative length along the coordinate points.

Returns array(N)

dists
distances between spatial steps along the tracked field line

Distance is returned in psi_n for dim = 1. In meters otherwise.

Returns

self._dists

impact_angle_cos()
Impact angle calculation - dot product of PFC norm and local magnetic field direction. Internally uses *incidence_angle_sin* function where *vecs* are replaced by the vector of the magnetic field.

Returns array of impact angles cosines

impact_angle_sin()
Impact angle calculation - dot product of PFC norm and local magnetic field direction. Internally uses *incidence_angle_sin* function where *vecs* are replaced by the vector of the magnetic field.

Returns array of impact angles sines

impact_angle_sin_pol_projection()
Impact angle calculation - dot product of PFC norm and local magnetic field direction poloidal projection only. Internally uses *incidence_angle_sin* function where *vecs* are replaced by the vector of the poloidal magnetic field (Bphi = 0).

Returns array of impact angles cosines

incidence_angle_cos(*vecs*)

Parameters **vecs** – array (3, N_vecs)

Returns array of cosines of angles of incidence

incidence_angle_sin(*vecs*)

Parameters **vecs** – array (3, N_vecs)

Returns array of sines of angles of incidence

intersection(*coords2*, *dim=None*)
input: 2 sets of coordinates crossection of two lines (2 sets of coordinates)

Parameters **dim** – reduce number of dimension in which is the intersection searched

Returns

length
Total length along the coordinate points.

Returns length in meters

line_integral(*func*, *method='sum'*)
func = /oint F(x,y) dl :param func: self - func(X, Y), Union[ndarray, int, float] or function values or 2D spline :param method: str, [‘sum’, ‘trapz’, ‘simps’] :return:

mesh()

normal_vector()
Calculate limiter normal vector with fw input directly from eq class

Parameters **first_wall** – interpolated first wall

Returns array (3, N_vecs) of limiter elements normals of the same

phi
Toroidal angle.

Returns Toroidal angle.

plot(*ax=None*, ***kwargs*)

Parameters

- **ax** – Axis to which will be plotted. Default is plt.gca()
- **kwargs** – Arguments forwarded to matplotlib plot function.

Returns

pol_projection_impact_angle_cos()
Impact angle calculation - dot product of PFC norm and local magnetic field direction poloidal projection only. Internally uses *incidence_angle_sin* function where *vecs* are replaced by the vector of the poloidal magnetic field (Bphi = 0).

Returns array of impact angles cosines

psi

psi_n

r

r_mid
Minor radius on the outer (magnetic) midplane. Minor radius is distance from magnetic axis.

Returns Minor radius mapped on the outer midplane.

resample(*multiple=None*)
Return new, resampled instance of *pleque.Coordinates*

Parameters **multiple** – int, use multiple to multiply number of points.

Returns pleque.Coordinates

resample2(*npoints*)
Implicit spline curve interpolation for the limiter, number of points must be specified

Parameters

- **coords** – instance of coordinates object
- **npoints** – int - number of points of the result

rho

theta

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

A

B

C

D

E

F

G

I

J

L

M

N

O

P

Q

R

S

T

V

X

Y

Z