FreeGS Documentation

Release 0.2.0

Ben Dudson

Feb 13, 2023

Contents:

CHAPTER 1

Creating equilibria

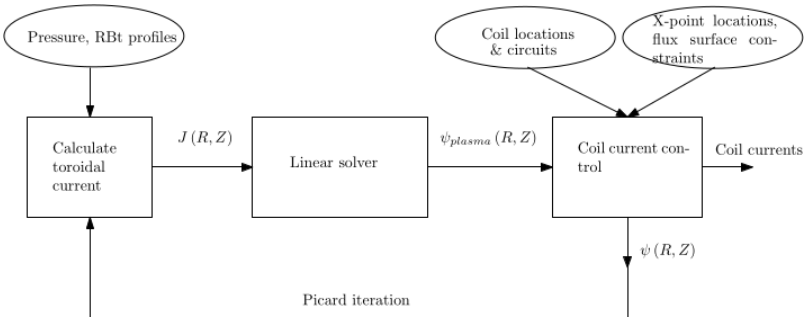
To generate a Grad-shafranov solution from scratch FreeGS needs some physical parameters:

1. The locations of the coils
2. Plasma profiles (typically pressure and a current function) used to calculate toroidal current density J_ϕ
3. The desired locations of X-points, and constraints on the shape of the plasma.

and some numerical parameters:

1. The domain in which we want to calculate the solution
2. The methods to be used to solve the equations

These inputs are combined in a nonlinear Picard iteration with the main components shown in the figure below.



1.1 Tokamak coils, circuits and solenoid

Example 1 (01-freeboundary.py) creates a simple lower single null plasma from scratch. First import the freegs library

```
import freegs
```

then create a tokamak, which specifies the location of the coils. In this example this is done using

```
tokamak = freegs.machine.TestTokamak()
```

which creates an example machine with four poloidal field coils (two for the vertical field, and two for the radial field). To define a custom machine, first create a list of coils:

```
from freegs.machine import Coil

coils = [("P1L", Coil(1.0, -1.1)),
         ("P1U", Coil(1.0, 1.1)),
         ("P2L", Coil(1.75, -0.6)),
         ("P2U", Coil(1.75, 0.6))]
```

Each tuple in the list defines the name of the coil (e.g. `"P1L"`), then the corresponding object (e.g. `Coil(1.0, -1.1)`). Here `Coil(R, Z)` specifies the R (major radius) and Z (height) location of the coil in meters.

Create a tokamak by passing the list of coils to `Machine`:

```
tokamak = freegs.machine.Machine(coils)
```

1.1.1 Coil current control

By default all coils can be controlled by the feedback system, but it may be that you want to fix the current in some of the coils. This can be done by turning off control and setting the current:

```
Coil(1.0, -1.1, control=False, current=50000.)
```

where the current is in Amps, and is for a coil with a single turn. Setting `control=False` removes the coil from feedback control.

1.1.2 Shaped coils

The `ShapedCoil` class models a coil with uniform current density across a specified shape (a polygon). This is done by splitting the polygon into triangles (using the ear clipping method) and then using Gaussian quadrature to integrate over each triangle (by default using 6 points per triangle). The shape of the coil should be specified as a list of `(R,Z)` points:

```
from freegs.machine import ShapedCoil

coils = [("P1L", ShapedCoil([(0.95, -1.15), (0.95, -1.05), (1.05, -1.05), (1.05, -1.
↪15)])),
         ...]
```

The above would create a square coil. The number of turns and the current in the circuit can be specified:

```
coils = [("P1L", ShapedCoil([(0.95, -1.15), (0.95, -1.05), (1.05, -1.05), (1.05, -1.
↪15)],
                            current = 1e3, turns=20)),
         ...]
```

The number of turns doesn't affect the integration, but the total current in the coil block is set to `current * turns`.

1.1.3 Multi strand coils

The `MultiCoil` class provides a way to model a coil block with turns in specified locations. For large numbers of turns this is usually more convenient than creating multiple `Coil` objects. A `MultiCoil` can be used anywhere a `Coil` object would be:

```
from freegs.machine import MultiCoil

coils = [ ("P2", MultiCoil([0.95, 0.95, 1.05, 1.05], [1.15, 1.05, 1.05, 1.15])) ]
```

For coils which are wired together as pairs, mirrored in Z, the `mirror=True` keyword can be given. The `polarity` keyword then sets the relative sign of the current in the original and mirrored coil.

1.1.4 Coil circuits

Usually not all coils in a tokamak are independently powered, but several coils may be connected to the same power supply. This is handled in FreeGS using `Circuit` objects, which consist of several coils. For example:

```
from freegs.machine import Circuit

Circuit( [("P2U", Coil(0.49,  1.76), 1.0),
          ("P2L", Coil(0.49, -1.76), 1.0)] )
```

This creates a `Circuit` by passing a list of tuples. Each tuple defines the coil name, the `Coil` object (with R,Z location), and a current multiplier. In this case the current multiplier is `1.0` for both coils, so the same current will flow in both coils. Alternatively coils may be wired in opposite directions:

```
Circuit( [("P6U", Coil(1.5,  0.9), 1.0),
          ("P6L", Coil(1.5, -0.9), -1.0)] )
```

so the current in coil "P6L" is in the opposite direction, but same magnitude, as the current in coil "P6U".

As with coils, circuits by default are controlled by the feedback system, and can be fixed by setting `control=False` and specifying a current.

1.1.5 Solenoid

Tokamaks typically operate with Ohmic current drive using a central solenoid. Flux leakage from this solenoid can modify the equilibrum, particularly the locations of the strike points. Solenoids are represented in FreeGS by a set of poiloidal coils:

```
from freegs.machine import Solenoid

solenoid = Solenoid(0.15, -1.4, 1.4, 100)
```

which defines the radius of the solenoid in meters (0.15m here), the lower and upper limits in Z (vertical position, here $\pm$ 1.4 m), and the number of poloidal coils to be used. These poloidal coils will be equally spaced between the lower and upper Z limits.

As with `Coil` and `Circuit`, solenoids can be removed from feedback control by setting `control=False` and specifying a fixed current.

1.1.6 Mega-Amp Spherical Tokamak

As an example, the definition of the Mega-Amp Spherical Tokamak (MAST) coilset is given in the `freegs.machine.MAST_sym()` function:

```
coils = [("P2", Circuit( [("P2U", Coil(0.49,  1.76), 1.0),
                          ("P2L", Coil(0.49, -1.76),1.0)] ))
         ,("P3", Circuit( [("P3U", Coil(1.1,  1.1), 1.0),
                          ("P3L", Coil(1.1, -1.1), 1.0)] ))
         ,("P4", Circuit( [("P4U", Coil(1.51,  1.095), 1.0),
                          ("P4L", Coil(1.51, -1.095), 1.0)] ))
         ,("P5", Circuit( [("P5U", Coil(1.66,  0.52), 1.0),
                          ("P5L", Coil(1.66, -0.52), 1.0)] ))
         ,("P6", Circuit( [("P6U", Coil(1.5,  0.9), 1.0),
                           ("P6L", Coil(1.5, -0.9), -1.0)] ))
         ,("P1", Solenoid(0.15, -1.45, 1.45, 100))
        ]

 tokamak = freegs.machine.Machine(coils)
```

This uses circuits "P2" to "P5" connecting pairs of upper and lower coils in series. Circuit "P6" has its coils connected in opposite directions, so is used for vertical position control. Finally "P1" is the central solenoid. Here all circuits and solenoid are under position feedback control.

1.2 Machine walls (limiters)

The internal walls of the machine are specified by a polygon in R-Z i.e. an ordered list of (R,Z) points which form a closed boundary. These are stored in a `Wall` object:

```
from freegs.machine import Wall

wall = Wall([ 0.75, 0.75,  1.5,  1.8,   1.8,   1.5],   # R
            [-0.85, 0.85, 0.85, 0.25, -0.25, -0.85])   # Z
```

The wall can then be specified when creating a machine:

```
tokamak = freegs.machine.Machine(coils, wall)
```

or an existing machine can be modified:

```
tokamak.wall = wall
```

Note that the location of these walls does not currently affect the equilibrium, but is used by some diagnostics, and is written to output files such as EQDSK format.

1.3 Sensors

Rogowski (Rog), Poloidal Field (BP), and Flux Loop (FL) sensors can be added to the machine.

Each individual sensor inherits from the Sensor class, which have inputs as follows:

```
class Sensor:
    def __init__(self, R, Z, name=None, weight=1, status=True, measurement=None):
```

R and Z take the input for the position of the sensors.

The weight is an attribute that is useful to consider with reconstruction on real data, can be thought of as the inverse of the sensor measurement uncertainty. Will discuss more in future chapters.

The status of a sensor determines whether it is turned on or not, and therefore whether it will take a measurement when `takeMeasurement` method of machine is called.

A list of sensors can be specified when creating a machine:

tokamak = freegs.machine.Machine(coils, wall, sensors)

1.3.1 Rogowski Sensors

Rog sensors measure the total current within their shape. The R-Z input is the same as the `Wall` class, taking an ordered list of (R,Z) points which form a closed boundary.

1.3.2 BP Sensors

BP sensors measure the poloidal field at a given angle. They are effectively 'point' sensors so single values for R and Z are given, aswell as a value for theta (angle of measurement)

1.3.3 FL Sensors

FL sensors measure the flux at an R,Z position. They are also point sensors, so need a single value for R and Z input.

Adding each one of these sensors to a machine when creating can be done as follows:

```
sensors = [RogowskiSensor(R = [ 0.75, 0.75,  1.5,  1.8,   1.8,   1.5],
                          Z = [-0.85, 0.85, 0.85, 0.25, -0.25, -0.85]),
           PoloidalFieldSensor(R = 0.75, Z = 1, theta = 0.9),
           FluxLoopSensor(R = 1.8, Z = 0.2)]

tokamak = Machine(coils, wall, sensors)
```

1.4 Equilibrium and plasma domain

Having created a tokamak, an `Equilibrium` object can be created. This represents the plasma solution, and contains the tokamak with the coil currents.

```
eq = freegs.Equilibrium(tokamak=tokamak,
                        Rmin=0.1, Rmax=2.0,     # Radial domain
                        Zmin=-1.0, Zmax=1.0,    # Height range
                        nx=65, ny=65)           # Number of grid points
```

In addition to the tokamak `Machine` object, this must be given the range of major radius R and height Z (in meters), along with the radial (x) and vertical (y) resolution. This resolution must be greater than 3, and is typically a power of 2 + 1 ($2^n + 1$) for efficiency, but does not need to be.

1.5 Boundaries

The boundary conditions to be applied are set when an Equilibrium object is created, since this forms part of the specification of the domain. By default a free boundary condition is set, using an accurate but inefficient method which integrates the Greens function over the domain. For every point $(\mathbf{R_b}, \mathbf{Z_b})$ on the boundary the flux is calculated using

$$\psi\left(R_b, Z_b\right) = \iint G(R, Z; R_b, Z_b) J_\phi\left(R, Z\right) dRdZ$$

where G is the Greens function.

An alternative method, which scales much better to large grid sizes, is von Hagenow's method. To use this, specify the `freeBoundaryHagenow` boundary function:

```
eq = freegs.Equilibrium(tokamak=tokamak,
                        Rmin=0.1, Rmax=2.0,     # Radial domain
                        Zmin=-1.0, Zmax=1.0,    # Height range
                        nx=65, ny=65,           # Number of grid points
                        boundary=freegs.boundary.freeBoundaryHagenow)
```

Alternatively for simple tests the `fixedBoundary` function sets the poloidal flux to zero on the computational boundary.

1.5.1 Conducting walls

To specify a conducting wall on which the poloidal flux is fixed, so that there is a skin current on the wall, a series of coils can be used. The current in each coil is set using the feedback controller, to satisfy a fixed poloidal flux constraint.

For the full example code, see (and try running) `09-metal-wall.py`.

First create an array of R,Z locations, here called `Rwalls` and `Zwalls`. For example a circular wall:

```
R0 = 1.0      # Middle of the circle
rwall = 0.5   # Radius of the circular wall

npoints = 200 # Number of points on the wall

# Poloidal angles
thetas = np.linspace(0, 2*np.pi, npoints, endpoint=False)

# Points on the wall
Rwalls = R0 + rwall * np.cos(thetas)
Zwalls = rwall * np.sin(thetas)
```

Then create a set of coils, one at each of these locations:

```
coils = [ ("wall_"+str(theta),     # Label
           freegs.machine.Coil(R, Z)) # Coil at (R,Z)
          for theta, R, Z in zip(thetas, Rwalls, Zwalls) ]
```

The label doesn't have to be unique , but having unique names makes referring to them later easier. The tokamak can then be created:

```
tokamak = freegs.machine.Machine(coils)
```

The next part is to control the currents in the coils using fixed poloidal flux constraints:

```
psivals = [ (R, Z, 0.0) for R, Z in zip(Rwalls, Zwalls) ]
```

This is a list of `(R, Z, value)` tuples, which specify that the poloidal flux should be fixed to zero (in this case) at the given `(R,Z)` location. The control system is then created:

```
constrain = freegs.control.constrain(psivals=psivals)
```

The final modification to the usual solve is that we can specify a poloidal flux for the plasma boundary:

```
freegs.solve(eq,          # The equilibrium to adjust
             profiles,    # The toroidal current profile function
             constrain,   # Constraint function to set coil currents
             psi_bndry=0.0)  # Because no X-points, specify the separatrix psi
```

If `psi_bndry` is set then this overrides the usual process, which uses the innermost X-point to set the plasma boundary psi. In this case there are some X-points between coils, but its more reliable to set the boundary like this.

1.6 Plasma profiles

The plasma profiles, such as pressure or safety factor, are used to determine the toroidal current J_ϕ:

$$J_\phi\left(R,Z\right) = R\frac{\partial p\left(\psi\right)}{\partial\psi} + \frac{f\left(\psi\right)}{R\mu_0}\frac{\partial f\left(\psi\right)}{\partial\psi}$$

where the flux function $p\left(\psi\right)$ is the plasma pressure (in Pascals), and $f\left(\psi\right) = RB_\phi$ is the poloidal current function. Classes and functions to handle these profiles are in `freegs.jtor`

1.6.1 Constrain pressure and current

One of the most intuitive methods is to fix the shape of the plasma profiles, and adjust them to fix the pressure on the magnetic axis and total plasma current. To do this, create a `ConstrainPaxisIp` profile object:

```
profiles = freegs.jtor.ConstrainPaxisIp(1e4, # Pressure on axis [Pa]
                                        1e6, # Plasma current [Amps]
                                        1.0) # Vacuum f=R*Bt
```

This sets the toroidal current to:

$$J_\phi = L\left[\beta_0 R + \left(1-\beta_0\right)/R\right]\left(1-\psi_n^{\alpha_m}\right)^{\alpha_n}$$

where ψ_n is the normalised poloidal flux, 0 on the magnetic axis and 1 on the plasma boundary/separatrix. The constants which determine the profile shapes are $\alpha_m = 1$ and $\alpha_n = 2$. These can be changed by specifying in the initialisation of `ConstrainPaxisIp`.

The values of L and β_0 are determined from the constraints: The pressure on axis is given by integrating the pressure gradient flux function

$$p_{axis} = -L\beta_0 R\int_{axis}^{boundary}\left(1-\psi_n^{\alpha_m}\right)^{\alpha_n}\,d\psi$$

The total toroidal plasma current is calculated by integrating the toroidal current function over the 2D domain:

$$I_p = L\beta_0\iint R\left(1-\psi_n^{\alpha_m}\right)^{\alpha_n}\,dRdZ + L\left(1-\beta_0\right)\iint\frac{1}{R}\left(1-\psi_n^{\alpha_m}\right)^{\alpha_n}\,dRdZ$$

The integrals in these two constraints are done numerically, and then rearranged to get L and β_0.

1.6.2 Constrain poloidal beta and current

This is a variation which replaces the constraint on pressure with a constraint on poloidal beta:

$$\beta_p = \frac{8\pi}{\mu_0}\frac{1}{I_p^2}\iint p\left(\psi\right) dRdZ$$

This is the method used in Y.M.Jeon 2015, on which the profile choices here are based.

```
profiles = freegs.jtor.ConstrainBetapIp(0.5, # Poloidal beta
                                        1e6, # Plasma current [Amps]
                                        1.0) # Vacuum f=R*Bt
```

By integrating over the plasma domain and combining the constraints on poloidal beta and plasma current, the values of L and β_0 are found.

1.7 Feedback and shape control

To determine the currents in the coils, the shape and position of the plasma needs to be constrained. In addition, diverted tokamak plasmas are inherently vertically unstable, and need vertical position feedback to maintain a stationary equilibrium. If vertical position is not constrained, then free boundary equilibrium solvers can also become vertically unstable. A typical symptom is that each nonlinear iteration of the solver results in a slightly shifted or smaller plasma, until the plasma hits the boundary, disappears, or forms unphysical shapes causing the solver to fail.

Currently the following kinds of constraints are implemented:

- X-point constraints adjust the coil currents so that X-points (nulls in the poloidal field) are formed at the locations requested.
- Isoflux constraints adjust the coil currents so that the two locations specified have the same poloidal flux. This usually means they are on the same flux surface, but not necessarily.
- Psi value constraints, which adjust the coil currents so that given locations have the specified flux.

As an example, the following code creates a feedback control with two X-point constraints and one isoflux constraint:

```
xpoints = [(1.1, -0.6),    # (R,Z) locations of X-points
           (1.1, 0.8)]

isoflux = [(1.1,-0.6, 1.1,0.6)] # (R1,Z1, R2,Z2) pairs

constrain = freegs.control.constrain(xpoints=xpoints, isoflux=isoflux)
```

The control system determines the currents in the coils which are under feedback control, using the given constraints. There may be more unknown coil currents than constraints, or more constraints than coil currents. There may therefore be either no solution or many solutions to the constraint problem. Here Tikhonov regularisation is used to produce a unique solution and penalise large coil currents.

1.8 Solving

To solve the Grad-Shafranov equation to find the free boundary solution, call `freegs.solve`:

```
freegs.solve(eq,          # The equilibrium to adjust
             profiles,    # The toroidal current profile
             constrain)   # Feedback control
```

This call modifies the input equilibrium (eq), finding a solution based on the given plasma profiles and shape control.

The Grad-Shafranov equation is nonlinear, and is solved using Picard iteration. This consists of calculating the toroidal current J_ϕ given the poloidal flux $\psi(R, Z)$, then solving a linear elliptic equation to calculate the poloidal flux from the toroidal current. This loop is repeated until a given relative tolerance is achieved:

$$\texttt{rtol} = \frac{\text{change in psi}}{\max(\psi) - \min(\psi)}$$

To see how the solution is evolving at each nonlinear iteration, for example to diagnose a failing solve, set `show=True` in the solve call. To add a delay between iterations set `pause=2.0` using the desired delay in seconds.

1.8.1 Inner linear solver

To calculate the poloidal flux given the toroidal current, an elliptic equation must be solved. To do this a multigrid scheme is implemented, which uses Jacobi iterations combined with SciPy's sparse matrix direct solvers at the coarsest level.

By default the multigrid is not used, and SciPy's direct solver is used for the full grid. This is because for typical grid resolutions (65 by 65) this has been found to be fastest. The multigrid method will however scale efficiently to larger grid sizes.

The easiest way to adjust the solver settings is to call the Equilibrium method `setSolverVcycle`. For example

```
eq.setSolverVcycle(nlevels = 4, ncycle = 2, niter = 10, direct=True)
```

This specifies that four levels of grid resolution should be used, including the original. In order to be able to coarsen (restrict) a grid, the number of points in both R and Z dimensions should be an odd number. This is one reason why grid sizes are usually $2^n + 1$; it allows the maximum number of multigrid levels.

The number of V-cycles (finest -> coarsest -> finest) is given by `ncycle`. At each level of refinement the number of Jacobi iterations to perform before restriction and again after interpolation is `niter`. At the coarsest level of refinement the default is to use a direct (sparse) solver.

Some experimentation is needed to find the optimium settings for a given problem.

CHAPTER 2

Input and Output

A standard format for storing tokamak equilibrium data is G-EQDSK which contains the poloidal flux in (R,Z) and 1D profiles of pressure, $f = RB_\phi$, safety factor q, and other quantities related to the Grad-Shafranov solution. The G-EQDSK format does not however have a standard for specifying the location of, and currents in, the poloidal field coils. This makes writing G-EQDSK files quite straightforward, but reading them more challenging, as these coil currents must be inferred.

The implementation of the file input and output is divided into a high level interface in `freegs.geqdsk` and a low level interface in `freegs._geqdsk`. The high level interface handles `Equilibrium` objects, whilst the low level interface handles simple dictionaries.

2.1 Writing G-EQDSK files

Import the `geqdsk` module from `freegs`, passing an `Equilibrium` object and a file handle:

```
from freegs import geqdsk

with open("lsn.geqdsk", "w") as f:
  geqdsk.write(eq, f)
```

2.2 Reading G-EQDSK files

This is complicated by the need to infer the currents in the coils. To do this the locations of the coils need to be specified. An example is `02-read-geqdsk.py` which reads a file produced by `01-freeboundary.py`. First create a machine object which specifies the location of the coils

```
from freegs import machine
tokamak = machine.TestTokamak()
```

Reading the file then consists of

```
from freegs import geqdsk
with open("lsn.geqdsk") as f:
  eq = geqdsk.read(f, tokamak, show=True)
```

This `read` function has the following stages:

1. Reads the plasma state from the file into an Equilibrium object
2. Uses the control system to find starting values for the coil currents, keeping the plasma boundary and X-point locations fixed
3. Runs the Grad-Shafranov picard solver, keeping profiles and boundary shape fixed. This adjusts the plasma solution and coil currents to find a self-consistent solution.

The `show` optional parameter displays a plot of the equilibrium, and shows the stages in the Grad-shafranov solve.

Some options are:

1. *domain = (Rmin, Rmax, Zmin, Zmax)* which can be used to specify the (R,Z) domain to solve on. This is useful for reading inputs from fixed boundary codes like SCENE, where the X-point(s) may lie outside the domain. The coil current feedback needs to find where the edge of the plasma is, so needs to find an X-point to work correctly. The grid spacing is never allowed to increase, so this may result in an increase in the number of grid points.
2. *blend* is a number between 0 and 1, and is used in the nonlinear Picard iteration. It determines what fraction of the previous solution is used in the next solution. The default is 0, so no blending is done. Adding blending (e.g. 0.5, 0.7) usually slows convergence, but can stabilise oscillating or unstable solutions.
3. *fit_sol* is *False* by default, so only points inside the separatrix are used when fitting the coil currents to match the input poloidal flux. This is useful in cases like reading SCENE inputs, where the poloidal flux in the Scrape-Off Layer (SOL) is not valid in the input. Setting *fit_sol=True* causes the whole input domain to be used in the fitting. Currently this is NOT compatible with setting a domain. It is useful when the locations of strike points need to be matched, and may better constrain coil currents for free boundary inputs (e.g. EFIT).

2.2.1 Specifying coil currents

A feedback control system is used to keep the plasma boundary and X-point locations fixed whilst adjusting the coil currents. If additional information about coil currents is available, then this can be used to fix some or all of the coil currents.

To see a list of the coils available:

```
print(tokamak.coils)

[('P1L', Coil(R=1.0,Z=-1.1,current=0.0,control=True)),
('P1U', Coil(R=1.0,Z=1.1,current=0.0,control=True)),
('P2L', Coil(R=1.75,Z=-0.6,current=0.0,control=True)),
('P2U', Coil(R=1.75,Z=0.6,current=0.0,control=True))]
```

Before calling `geqdsk.read`, specify the coil currents in the `tokamak` object:

```
tokamak["P1L"].current = 5e4  # Amp-turns
```

This will give the control system a starting value for the coil currents, but since the coil is still under feedback control it may still be altered. To fix the current in the coil turn off control:

```
tokamak["P1L"].control = False  # No feedback control (fixed current)
```

2.3 Writing DivGeo files

These are used as input to the SOLPS grid generator. They contain a subset of what's in G-EQDSK files, with no plasma pressure or current profiles.

To convert a G-EQDSK file directly to DivGeo without any solves, and without needing to know the coil locations, use the low-level routines:

```
from freeqdsk import geqdsk
from freegs import _divgeo

with open("input.geqdsk") as f:
  data = geqdsk.read(f)

with open("output.equ", "w") as f:
  _divgeo.write(data, f)
```

FreeGS equilibria objects can also be written to DivGeo files using the *divgeo* module:

```
from freegs import geqdsk
from freegs import divgeo
from freegs.machine import TestTokamak

# Read a G-EQDSK file
with open("lsn.geqdsk") as f:
  eq = geqdsk.read(f, TestTokamak(), show=True)

# Modify the equilbrium...

# Save to DivGeo
with open("lsn.equ", "w") as f:
  divgeo.write(eq, f)
```

CHAPTER 3

Diagnostics

Once an equilibrium has been generated (see **creating_equilibria_**) there are routines for diagnosing and calculating derived quantities. Here the `Equilibrium` object is assumed to be called `eq` and the `Tokamak` object called `tokamak`.

3.1 Safety factor, q

The safety factor q at a given normalised poloidal flux ψ_{norm} (0 on the magnetic axis, 1 at the separatrix) can be calculated using the `q(psinorm)` function:

```
eq.q(0.9)  # safety factor at psi_norm = 0.9
```

Note that calculating q on either the magnetic axis or separatrix is problematic, so values calculated at $\psi_{norm} = 0$ and $\psi_{norm} = 1$ are likely to be inaccurate.

This function can be used to print the safety factor on a set of flux surfaces:

```
print("\nSafety factor:\n\tpsi \t q")
for psi in [0.01, 0.9, 0.95]:
  print("\t{:.2f}\t{:.2f}".format(psi, eq.q(psi)))
```

If no value is given for the normalised psi, then a uniform array of values between 0 and 1 is generated (not including the end points). In this case both the values of normalised psi and the values of q are returned:

```
psinorm, q = eq.q()
```

which can be used to make a plot of the safety factor:

```
import matplotlib.pyplot as plt

plt.plot(*eq.q())
plt.xlabel(r"Normalised $\psi$")
```

(continues on next page)

(continued from previous page)

```
plt.ylabel(r"Safety factor $q$")
plt.show()
```

3.2 Poloidal beta

The poloidal beta β_p is given by:

```
betap = eq.poloidalBeta()
```

This is calculated using the expression

$$\beta_p = \frac{8\pi}{\mu_0}\frac{1}{I_p^2}\iint p\left(\psi\right) dRdZ$$

i.e. the same calculation as is done in the poloidal beta constraint **constrain_betap_ip_**.

3.3 Plasma pressure

The pressure at a specified normalised psi is:

```
p = eq.pressure(0.0)  # Pressure on axis
```

3.4 Separatrix location

A set of points on the separatrix, measured in meters:

```
RZ = eq.separatrix()

R = RZ[:,0]
Z = RZ[:,1]

import matplotlib.pyplot as plt
plt.plot(R,Z)
```

3.5 Currents in the coils

The coil objects can be accessed and their currents queried. The current in a coil named “P1L” is given by:

```
eq.tokamak["P1L"].current
```

The currents in all coils can be printed using:

```
tokamak.printCurrents()
```

which is the same as:

```
for label, coil in eq.tokamak.coils:
  print(label + " : " + str(coil))
```

3.6 Forces on the coils

The forces on all poloidal field coils can be calculated and returned as a dictionary:

```
eq.getForces()
```

or formatted and printed:

```
eq.printForces()
```

These forces on the poloidal coils are due to a combination of:

- The magnetic fields due to other coils
- The magnetic field of the plasma
- A self (hoop) force due to the coil's own current

The self force is the most difficult to calculate, since the force depends on the cross-section of the coil. The formula used is from David A. Garren and James Chen (1998). For a circular current loop of radius R and minor radius a, the outward force per unit length is:

$$f = \frac{\mu_0 * I^2}{4\pi R} (ln(8 * R/a) - 1 + \xi_i/2)$$

where ξ_i is a constant which depends on the internal current distribution. For a constant, uniform current $\xi_i = 1/2$; for a rapidly varying surface current $\xi_i = 0$.

For the purposes of calculating this force the cross-section is assumed to be circular. The area can be set to a fixed value:

```
tokamak["P1L"].area = 0.01  # Area in m^2
```

where here "P1L" is the label of the coil. The default is to calculate the area using a limit on the maximum current density. A typical value chosen here for Nb3Sn superconductor is $3.5 \times 10^9 A/m^2$, taken from Kalsi (1986) .

This can be changed e.g:

```
from freegs import machine

tokamak["P1L"].area = machine.AreaCurrentLimit(1e9)
```

would set the current limit for coil "P1L" to 1e9 Amps per square meter.

3.7 Sensor Measurements

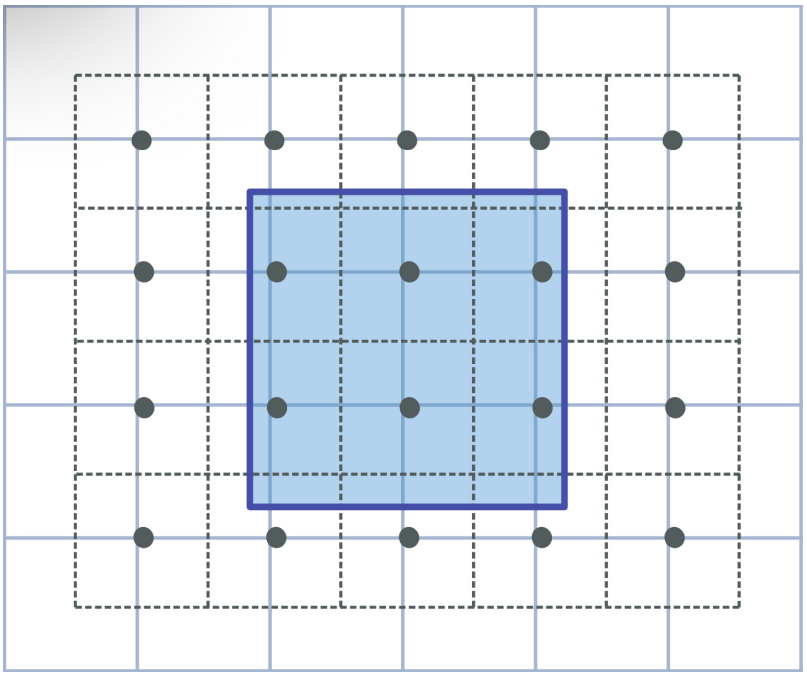
For a machine populated with sensors, to run the `get_Measure(equilibrium)` method of each sensor, `tokamak.takeMeasurments(equilibrium)` is used. The measurement atrribute of each sensor is then updated with the measured values. If no equilibrium object is passed, then the sensors will find the coil contribution (and in a future update, vessel current contribution) to each of the measurements.

To measure and print the values, the following method is used:

```
from freegs import machine
tokamak = machine.TestTokamakSensor()

tokamak.printMeasurements(equilibrium)
```

The Rogowski Coils uses a nearest neighbour interpolation method. The following diagram illustrates this. The points correspond to each grid point on the equilibrium grid. A shapely square object is created, centered around the point. The sensor calculates the intersection area of each square with the rog and multiplies it by the value of the current density at that point.



Both the BP and FL sensors use pre specified methods using interpolation in the machine and equilibrium classes.

3.8 Field line connection length

Example: `10-mastu-connection.py`. Requires the file `mast-upgrade.geqdsk` which is created by running `08-mast-upgrade.py`.

To calculate the distance along magnetic field lines from the outboard midplane to the walls in an equilibrium `eq`, the simplest way is:

```
from freegs import fieldtracer
forward, backward = fieldtracer.traceFieldLines(eq)
```

To also plot the field lines on top of the equilibrium:

```
axis = eq.plot(show=False)
forward, backward = fieldtracer.traceFieldLines(eq, axis=axis)
plt.show()
```

This will display the poloidal cross-section of the plasma, and plot field lines traced in both directions along the magnetic field from the outboard midplane.

To plot the distances along the magnetic field from midplane to target as a function of the starting radius:

```
plt.plot(forward.R[0,:], forward.length[-1,:], label="Forward")
plt.plot(backward.R[0,:], backward.length[-1,:], label="Backward")
plt.legend()
plt.xlabel("Starting major radius [m]")
plt.ylabel("Parallel connection length [m]")

plt.show()
```

Here `forward.R` and `forward.length` are 2D arrays, where the first index is the point along the magnetic field (0 = start, -1 = end), and the second index is the field line number. There is also `forward.Z` with the height in meters.

The output can be customised by passing keywords to `traceFieldLines`: `solwidth` sets the width of the starting region at the outboard midplane; `nlines` is the number of field lines to follow in each direction; `nturns` the number of times around the torus to follow the field; `npoints` is the number of points along each field line.

For more control over which field lines are followed, the `FieldTracer` class does the actual field line following:

```
from freegs import fieldtracer
import numpy as np

tracer = fieldtracer.FieldTracer(eq)

result = tracer.follow([1.35], [0.0], np.linspace(0.0, 2*np.pi, 20))
```

This follows a magnetic field in the direction of B, starting at $R = 1.35m$, $Z = 0$, outputting positions at 20 toroidal angles between 0 and 2pi i.e. one toroidal turn. The R and Z starting locations should be an array or list with the same shape.

The `result` is an array: The first index is the angle (size 20 here), and the last index has size 3 (R, Z, length). Between the first and last indices the result has the same shape as the R and Z starting positions. In the above code `result` has size `(20, 1, 3)`. To plot the field line on top of the equilibrium:

```
import matplotlib.pyplot as plt

eq.plot(show=False)
plt.plot(result[:,0,0], result[:,0,1])
plt.show()
```

The direction to follow along the field can be reversed by passing `backward=True` keyword to `tracer.follow`.

CHAPTER 4

Tests

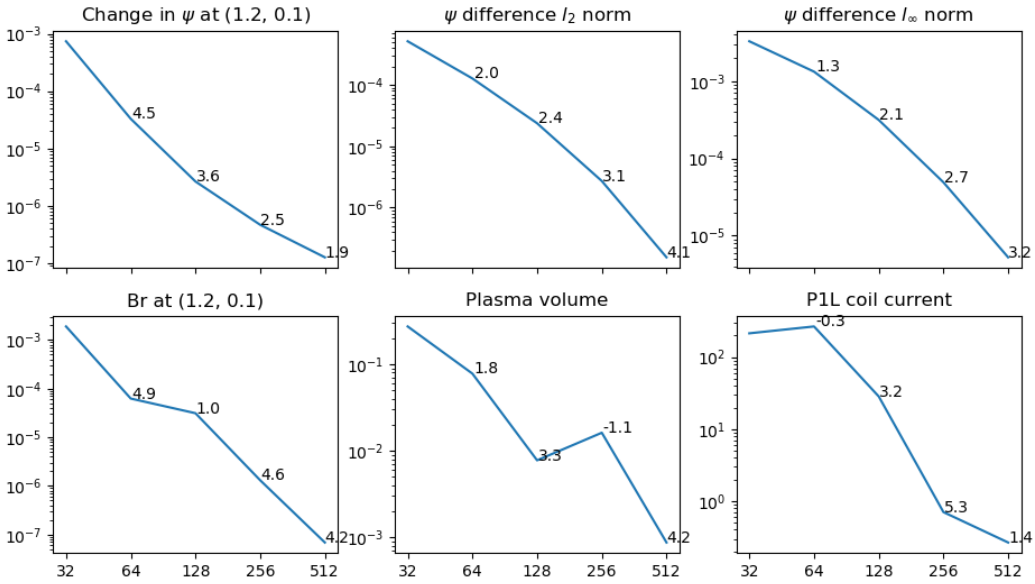
4.1 Unit tests

Unit testing is done with pytest. Run the tests in the `freegs` directory with:

```
pytest
```

4.2 Convergence test

The `test-convergence.py` script solves for the same plasma as the `01-freeboundary.py` example, with four poloidal field coils and an X-point. The change in results as the resolution is doubled is plotted as a function of grid resolution. This is therefore not testing convergence to a known solution, but is a check that the code converges to value, and indication of its convergence rate. Results are shown below, using the von Hagenow free boundary, and 4th-order solver for the poloidal flux ψ.



This indicates that in general convergence is between 2nd and 4th-order. The plasma volume is currently calculated by integrating over the poloidal cross-section, and could be improved by converting this to a surface integral.

CHAPTER 5

Optimisation

This is an experimental feature which is at an early stage of development. The aim is to enable equilibria to be automatically optimised. This has the following components:

1. Measures, a quantity which measures how "good" a solution is. Typically the aim is to minimise this quantity, so I suppose it's really a measure of how bad the solution is.
2. Controls, quantities which can be changed. These could be machine parameters such as coil locations, constraints like X-point location, or plasma profiles such as poloidal beta or plasma current.
3. An algorithm which modifies the controls and finds the best equilibrium according to the measure it's given. At the moment the method used is Differential Evolution.

5.1 Differential Evolution

Differential Evolution is a type of stochastic search, similar to Genetic Algorithms, generally well suited to problems involving continuously varying parameters.

The implementation of the algorithm is in freegs.optimiser. It is generic, in that it operates on objects but does not need to know any details of what those objects are. To modify objects a list of `controls` are passed to the optimiser, each of which can set and get a value. To score each object a `measure` function is needed, which takes an object as input and returns a value. The optimiser works to minimise this value.

An example which uses the optimisation method is in the `freegs` directory. This optimises a quadratic in 2D rather than tokamak equilibria. 100 generations are run, with 10 solutions (sometimes called agents) in each generation. Run this example with the command:

```
python test_optimiser.py
```

This should produce the figure below. The red point is the best solution at each generation; black points are the other points in that generation. Faded colors (light red, grey) are used to show previous generations. It can be seen that the points are clustered around the starting solution, as the agents spread out, and then around the solution as the agents converge to the minimum.

5.2 Optimising tokamak equilibria

The code specific to FreeGS optimisation is in freegs.optimise. This includes controls which modify aspects of the tokamak or equilibrium:

Control	Description
CoilRadius(name [, min, max])	Modify coil radius, given name and optional limits
CoilHeight(name [, min, max])	Modify coil height

Measures which can be used by themselves or combined to specify the quantities which should be optimised:

Measure function	Description
max_abs_coil_current	Maximum current in any coil circuit
max_coil_force	Maximum force on any coil
no_wall_intersection	Prevent intersections of wall and LCFS

Each measure function takes an Equilibrium as input, and returns a value. These can be combined in a weighted sum using `optimise.weighted_sum`, or by passing your own function to `optimise`.

The example 11-optimise-coils.py uses the following code to reduce the maximum force on the coils, while avoiding wall intersections:

```
from freegs import optimise as opt

best_eq = opt.optimise(eq,  # Starting equilibrium
                       # List of controls
                       [opt.CoilRadius("P2U"),
                       opt.CoilRadius("P2L"), opt.CoilHeight("P2L")],
                       # The function to minimise
                       opt.weighted_sum(opt.max_coil_force, opt.no_wall_intersection),
                       N=10,  # Number of solutions in each generation
                       maxgen=20, # How many generations
                       monitor=opt.PlotMonitor())  # Plot the best in each generation
```

The monitor should be a callable (here it is an object of class `PlotMonitor`), which is called after each generation. This is used to update a plot showing the best solution in each generation, and save the figure to file.

#.. automodule:: freegs.equilibrium # :members:

CHAPTER 6

Indices and tables

- genindex
- modindex
- search