Development of a Python code for constructing a field-aligned mesh

September 24, 2020

1 Introduction

Magnetic nuclear fusion is nowadays considered as one of the most promising candidate for electricity production in a clean and safe way. The most studied devices that aim to achieve nuclear fusion are called tokamaks and have been studied since the end of the second world war. In these devices, light nuclei (mostly hydrogen isotopes) are heated up to extremely high temperatures (hundreds of million of degrees) until a thermonuclear plasma is obtained. A plasma can be though of as a collection of electrons and ions, which is however globally neutral. Lorentz force can thus interact with the plasma particles (ions and electron) and one can therefore confine it in a bounded region of space using magnetic fields. The divergence-free condition on the magnetic field, $\nabla \cdot \boldsymbol{B} = 0$, imposes a restriction on the topology of the field lines: they must be either close, or extend to infinity. Fields of the latter type are used in the so-called *open configuration* machines (sometimes called linear machines), the best known of which are the *magnetic mirrors*. In such a configuration, there always exists a loss-cone, which must be plugged by various means. Closed magnetic field lines are necessarily confined in a bound region of space. But even in this case, the global topology of the field lines is not always simple, except when their total length is finite. In general, though, this is not the case. The field line may wander indefinitely through the bounded region in which it is confined, before closing on itself after an infinite journey. An inescapable consequence of this topology is the fact that such a line densely fills some region of the domain in which it is confined; that is, it passes arbitrarily close to any point of this region. The latter may be the whole domain or some three-dimensional subdomain; finally, the region densely covered by field lines may be a two-dimensional surface. A Theorem of topology guarantees that, if the magnetic field $\boldsymbol{B}$ never vanishes on this surface and, moreover, this surface does not have any edge, then it must be a toroid. This type of topology is used in the tokamaks.

2 A brief introduction to tokamaks

The tokamak concept was first proposed by the two Russian physicist Tamm and Andrei Sakharov in 1952. The name is a Russian acronym standing for toroidal chamber with magnetic coils. A schematic picture of a tokamak device is shown in figure 1 One of the characteristics features of the tokamak geometry is its symmetry in the toroidal direction. This gives

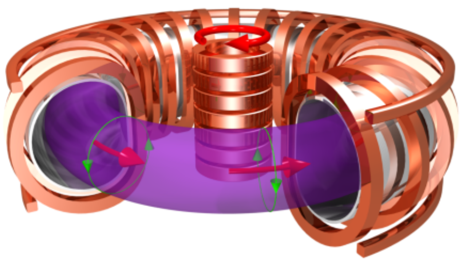

Figure 1: Schematic picture of a tokamak device. The plasma (shown in purple) is confined thanks to the combined used of two magnetic field components: the toroidal one (shown in red) and the so-called poloidal one (shown in green).

huge advantages in the modelling of these devices, which reduces the computational complexity moving from $3D$ simulations to $2D$ ones.

The confinement of the plasma is provided by the magnetic field structure outlined in figure 1. This confinement is however not perfect. Some of the plasma particles indeed escape from the confinement region and can interact with the solid walls of vacuum chamber. This gives rise to a multitude of phenomena which goes under the name of plasma-material interaction (PMI). PMI occurs mostly in the outermost part of the tokamak device. The plasma that resides in this region is called the **edge plasma** or **scrape-off layer**. This PMI should be limited as much as possible, since they limit the lifetime of the walls' component as well as the overall performance of the plasma. Study these interactions is therefore of great importance, particularly for future tokamak devices where the plasma conditions (density, temperature, etc.) are not known. To this end, linear plasma devices are of great interest. They are cost-effective and compact machines, characterised by a cylindrical vacuum vessel. Confinement in the radial direction is provided by a set of magnetic field coils, as shown in figure 2. Here we take as an example the linear plasma device GyM of ISTP-CNR in Milan.

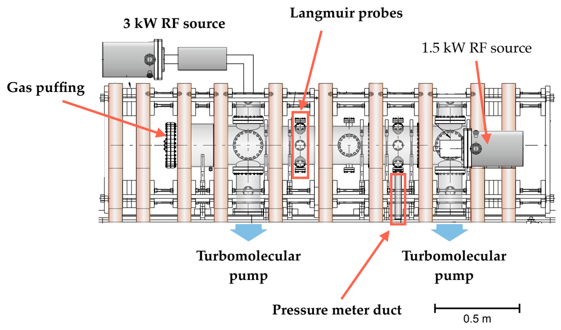

Figure 2: Schematic drawing of the linear plasma device GyM of ISTP. Shown in the figure are the vacuum vessel, the magnetic field coils, the turbomolecular pumps for the background vacuum and the radiofrequency source to generate and heat the plassma.

3 Edge plasma codes

Advanced computational models have been developed to interpret and predict the plasma behaviour in the edge region of present-day tokamaks. Edge plasma codes usually consists of two major modules: one for the plasma species (electron and ions) and one for neutral atoms and molecules. The former is usually based on a fluid description of the plasma, akin to the Navier-Stokes equation for a compressible fluids, with a proper closure (so-called Braginskii closure) for the transport coefficients (e.g., electron heat flux). The plasma fluid equations are generally solved using a finite volume method. The latter is instead a Monte Carlo code, which solves the Boltzmann equation. The two codes are coupled iteratively: the fluid code provide the (fixed) plasma background upon which Monte Carlo histories of neutral particles are computed. These latter then provide the source terms which appear in the fluid equations. The whole cycle just outlined is then repeated until convergence of the solution is obtained. The model that have been described above is implemented in the edge plasma code SOLPS (Scrape-Off Layer Plasma Simulation). It consists of two major part, namely B2.5 (the fluid code) and Eirene (the Monte Carlo code). One of the most important input to perform a simulation with this code suites is the generation of a computational grid.

4 Computational grid for edge plasma codes

We shall describe here only the B2.5 grid, since it is the only one relevant for the project. It consists of quadrilaterals and it must satisfy the requirement of being field-aligned. That is, two side of the quadrilaterals must align with the magnetic field lines. This is due to the fact that plasma transport is strongly anistoropical: it is fast along magnetic field lines and slow across. The grid should therefore account for these physical properties in order to minimise numerical diffusion. Thus, as a preliminary step for the construction of such a grid it is first necessary to reconstruct the magnetic equilibrium. Under the assumption of axisymmetry, the latter can be obtained as a solution of the so-called Grad-Shafranov equation:

$$\Delta^* \psi = -\mu_0 R J_\varphi \tag{4.1}$$

In this equation, Δ^* is the Stokes operator defined as

$$\Delta^* = R\partial_R \left(\frac{1}{R} \partial_R \psi \right) + \partial_Z^2 \psi, \tag{4.2}$$

μ_0 the vacuum magnetic permeability and J_φ the toroidal currents. These include contributions either coming from the plasma itself or those due to external coils. For a linear plasma device, the former can in general be neglected. The ψ function plays the role of a stream-function: its level set contains the magnetic field lines. Indeed, the latter can be simply compute as

$$B_R = \frac{1}{R} \frac{\partial \psi}{\partial Z} \tag{4.3}$$

$$B_Z = -\frac{1}{R} \frac{\partial \psi}{\partial R} \tag{4.4}$$

4.1 Solution of the Grad-Shafranov equation

Solution of equation (4.1) can be based on several approaches. One of the possibility involves the `b2mn.dat` function for the Stokes operator, that is the solution of (4.1) can be written as

$$\psi\left(R, Z\right) = \iint G\left(R, Z; R', Z'\right) J_\varphi\left(R', Z'\right) \mathrm{d}R' \mathrm{d}Z' \tag{4.5}$$

with

$$\begin{aligned} G\left(R, Z; R', Z'\right) &= \frac{\mu_0}{2\pi} \frac{\sqrt{RR'}}{k} \left[\left(2 - k^2\right) K\left(k\right) - \right. \\ &\quad \left. - 2E\left(k\right) \right] \\ k^2 &= \frac{4RR'}{\left(R + R'\right) + \left(Z - Z'\right)^2} \end{aligned}$$

In this expression, $K\left(k\right)$ and $E\left(k\right)$ are the elliptic integrals of the first and second kind, respectively. In the remaining of this section, we shall describe the Python code developed for the construction of the equilibrium magnetic field. The code rely on the Python library `numpy` and `scipy`, as it will be described in the following. For the computation of the integral in (4.1) knowledge of the position of the different coils and their current density is needed. Coils information are stored in the Python class `Coils`. The `__init__` method first ask for a .txt file where the geometrical information of the coils and the current flowing in each of the windings are stored. The method `ComputeCurrents` then computes the current density, simply dividing the current by the cross-sectional area (a rectangle) of each coil and storing the information in the variable `Jc`.

After information of the coils are know, one can then compute the equilibrium solving the Grad-Shafranov equation (4.1) or, equivalently, the integral form (4.5). This is done the class `Equilibrium`. The vacuum permeability (μ_0 in (4.1)) is defined as an hidden property, `self .__mu0`. The `__init__` method take as an input six parameters, `def __init__(self, r1, r2, z1, z2, Nr, Nz, Nc = 10)`, which define the extension of the square computational domain where the equilibrium is to be computed as well as the resolution (the number of points in the radial, `Nr`, and axial, `Nz`) direction). The seventh argument, `Nc=10` is optional and it is related to the discretisation of the square domain defining the cross-sectional area of each of the magnetic field coils.

The core of the computation occurs in the `ComputePsi` method. Three nested loop are used to compute the Green function for the GS operator, which is first initialised as a $4D$ numpy array. The first `for` loop ranges over the number of external magnetic field coils, `Ncoil`. The remaining two over the square domain, `Nc`×`Nc`, defining the coils extension. It should be noted that, with respect to a C-style code, the two nested loop ranging over the computational domain, `Nr`×`Nz`, are avoided, thanks to broadcasting. This allow for an extreme code speed up, the computation of a 105 × 105 equilibrium requiring only few seconds. The elliptic integrals of the first and second kind are computed with the `ellipk` and `ellipe` functions in `scipy`. Integration of equation 4.5 is then performed thanks to the trapeziodal method `trapz` in `numpy` and the result stored in the $3D$ numpy array `psi_coil`. Finally, the contribution from each of the external magnetic field coil is summed over and the result is stored into the $2D$ numpy array `psi`. Once the information of the ψ function is known over all the `Nr`× `Nz`, one can then compute the two component of the magnetic field B_R, B_Z using equations (4.3) and (4.4). This is numerically performed in the method `ComputeField` of the class `Equilibrium` using a standard finite difference scheme and the result is stored in the two $2D$ numpy array `Br,Bz`.

4.1.1 Code profiling

The performance of the code have been tested using the Python code profiler `cProfile`. Thanks to the profiler, it is possible to understand where the time spent by the code at each of the functions calls. For example, computation of a 105×105 equilibrium requires $\simeq 0.5$ s and essentially all of this time is spent in the `ComputeField` function. Half of this time is spent in the computation of the elliptic integrals in `ellipk` and `ellipe` . It is also interesting to compare what are the code performances by performing the computation using a C-style coding style, that is avoiding to use broadcasting but rather using two additional nested loop, as has been briefly described above. In this case, the computational time in-

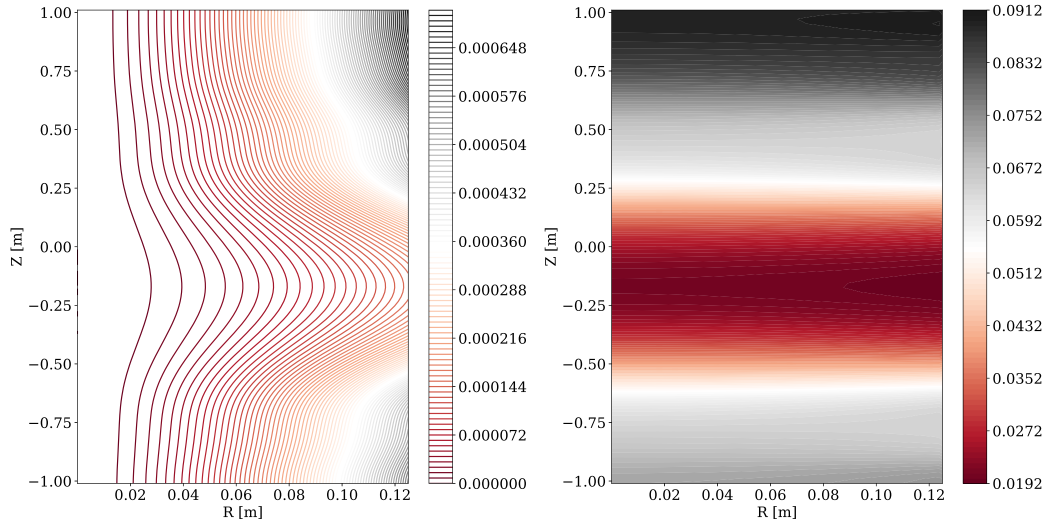

Figure 3: Example of a $2D$ equilibrium. The ψ function contour lines are shown on the left; the magnetic field strength, B, is shown on the right.

crease drastically to almost two minutes. Again, most of this time is spent in the computation of the ψ function. This C-type style of coding style have also been implemented in a Matlab code, showing equal performances between the two languages.

4.2 Plotting the equilibrium

Once the computation of the ψ function is performed, one can plot its contour line over the $2D$ domain where it is defined using the `PlotEquilibrium` class. It also allows to plot the contour lines of the total magnetic field, $B = \sqrt{B_R^2 + B_Z^2}$. The class takes as an input the `Equilibrium` object, as well as two additional optional arguments, `Ncont` and `Dim`, which specify the number of contours to be plotted and the dimension of the axes label, respectively. The Python package `matplotlib` and `pyplot` are used to this end.

As an example of the two classes which have been just described, we show here the computation of a $2D$ equilibrium defined by ten external field coils, two of which have been switched off. The resulting magnetic field is then similar to what is commonly known as a magnetic mirror and it is shown in figure 3. The contour lines of the ψ function are shown on the left, while the magnetic field intensity is reported on the right, both using a gray-scale map, `cmap=matplotlib.cm.RdGy`.

4.3 Construction of a field-aligned grid

Edge plasma codes are usually based on a finite volume scheme for the solutions of the fluid equations. The computational grid is then usually structured and made of quadrilaterals. However, it must also satisfy also the requirement of being field-aligned: that is, two of the sides of each quadrilateral must lie on magnetic field lines. Since the ψ function is a stream-function, then its contour lines contains the magnetic field vector $\boldsymbol{B}$. So the computation of the grid is strictly based on the knowledge of ψ.

The construction of this type of grid is performed in the class `Mesh`. The constructor method takes as input four quantities, namely the equilibrium (`Eq`), the number of grid points in the radial direction (`m`), the number of grid points in the axial direction (`n`) and the position of the vessel vacuum chamber (`Rlim`). Additionally, two hidden properties of the `Mesh` class are also defined, namely `Ncont` and `LowerPsiValue`, whose role will be described in the following. The constructor first computes the last magnetic field lines before intersection of the level-set of ψ with the vacuum vessel position (specified by `Rlim` occurs). This is done in the `ComputeLastPsi` method. Here a `numpy` array , `ContourLevels`, of dimension `Ncont` is first defined, ranging from `LowerPsiValue` and `psi.max()`. The contour lines of ψ are then computed and the result stored in `Cnt`. Then, an empty list `lines` is defined. A `for` loop cycles over the dimension of `ConourLevels` (i.e., `Ncont`). An inner `for` loop is then performed, which cycles over all the computed contour lines (`Cnt.collections[it].get_paths()`). The vertices ((x, y) coordinates) defying each level set are then appended to the list `lines`. Finally, the difference between the maximum of eac radial coordinate of these contour lines and the radial position of the vessel vacuum chamber is computed. If the difference is positive or equal to zero, the code prints out a message and the last value of the ψ function stored in `PsiLim`.

The core of the computation occurs in the `ComputeGrid` method. First, a `numpy` array `ContourLevels` having dimension `m` is defined, ranging from `LowerPsiValue` to `PsiLim`. Mesh refinement is needed at every the points where the plasma touches solid surfaces, since strong gradients exist there. To this end, a tanh function is used to satisfy this requirement. These points are then stored in a `numpy` array (`zQuery`), having dimension `n`, that is the number of grid points in the axial direction. The ψ function need then to be interpolated over these new grid points. This is done in the method `InterpolatePsi`, where a spline function (relying on the `scipy` library) is used to interpolate the values of ψ over the new grid points, `zQuery`. The result is then stored into a $2D$ `numpy` array `InterpPsi`, having dimensions `n`$\times$`m`. This array is then use, together with `zQuery` in order to build each of the quadrilateral cells that makes the computational grid. Small guard cells are also added and treated as guard cells for the boundary conditions. Finally, each of the cells' vertex are assembled in a `numpy` array `MESH`. The dimension of this array are code-specific. For SOLPS, the structure of the array is the following. The number of columns is fixed to fifteen. The first two columns are the cell indicator; that is, each of the grid cell is assigned a unique couple of numbers. The successive ten entries are the radial and axial position of the five vertices (including also the cells' center vertices) that make up the quadrilaterals. Finally, the remaining two entries specify the magnetic field components (radial and axial) at each of the quadrilateral cell center. Again, since the magnetic field is known on a rectangular grid, an interpolation procedure is first performed in the method `InterpolateField`. Here we use the `scipy` module and, in particular, the `RectBivariateSpline`.

Once computed, the mesh can then be plotted using the `PlotMesh` method. For the external coil structure of figure 3, we show in figure 4 the corresponding computational grid build according to the scheme outlined above. Finally, the mesh

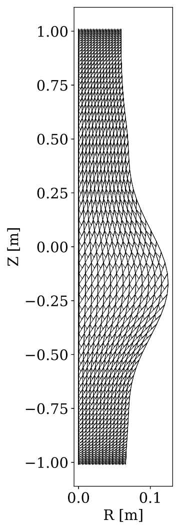

Figure 4: Computational grid for the magnetic equilibrium shown in figure 3.

can be written to an text file using the `WriteMesh` method.

5 Additional input for linear plasma device simulations

As already mentioned, linear plasma devices are characterised by a rather simple structure: a cylindrical vacuum vessel and a set of external magnetic field coils. The plasma in these devices can be generated according to different mechanism. One of the possibility rely on the interaction of electromagnetic waves and the plasma. This can lead to a resonant exchange of energy between the wave and the plasma species, which can be either electrons or ions. The type of resonances can depend upon several different parameters, such as the local plasma density or the direction between the electromagnetic wave propagation and the magnetic field. For simplicity, here we shall describe only the case where an electromagnetic wave interacts with the plasma and the local incidence angle between the magnetic field vector and the wave is 90°. In this case, three different type of resonances can occur.

$$\Omega_{\mathrm{EC}} = \frac{eB}{m_e} \tag{5.1}$$

$$\Omega_{\mathrm{UH}} = \sqrt{\omega_p^2 + \Omega_{\mathrm{EC}}^2} \tag{5.2}$$

$$\Omega_{\mathrm{LH}} = \sqrt{\frac{\omega_p^2 + \Omega_{IC}^2}{1 + \omega_p^2/\Omega_{EC}^2}} \tag{5.3}$$

which are the **electron cyclotron**, **upper-hybrid** and **lower hybrid** frequencies respectively. In these expressions, ω_p denotes the so-called plasma frequency which is defined as:

$$\omega_p = \sqrt{\frac{n_e e^2}{m_e \epsilon_0}} \tag{5.4}$$

which depends on the electron density, and Ω_{IC} is the ion cyclotron frequency (that is, the same of (5.1), with the ion mass replacing the electron mass). A resonance occurs whenever the electromagnetic wave frequency matches one of the above-reported resonances.

5.1 Plotting resonances on the computational mesh

Dedicated code have been developed to study this particular issue and, moreover, to compute the amount of power deposited into the plasma as well as its spatial distribution. In edge plasma codes, these type of contributions appears simply as additional external sources into the relevant plasma fluid equations. For example, assuming that the wave excite only the electron

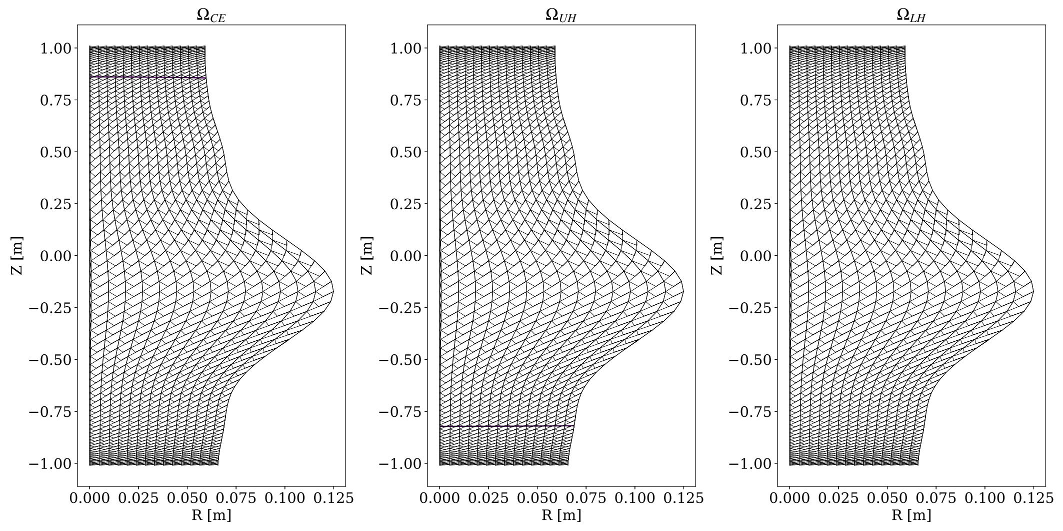

Figure 5: Electron cyclotron, upper hybrid and lower hybrid resonances plotted on the computational mesh of figure 4. The resonance location (a line) is plotted in purple.

population, then one can model the power delivered to them simply as an additional source term in the electron energy equation. A simple tool was developed in Python to this end and it is implemented in the class `Resonances` . The constructor takes as an input the following arguments: `def__init__(self, OmegaInj, Eq, Mesh, neFit)`. In this expression, `OmegaInj` denotes the frequency of the injected electromagnetic wave, `Eq` is the object representing the magnetic equilibrium, `Mesh` is the object representing the computational mesh, and `neFit` is the assumed distribution of the electron density within the plasma. The latter is indeed needed since the upper-hybrid frequency (cfr. (5.2)) depends also on the local plasma density. Computation of the resonances is performed in the `ComputeResonances` method. It simply implements the formulae that have been reported in the last paragraph, and return the corresponding frequency as $2D$ numpy arrays. These three resonances can then be plotted to see whether or not they can occur for the set of parameter considered. This plot can be performed either on the same square domain where the magnetic equilibrium is defined (`PlotResonances`) or, more conveniently, on the computational mesh (`PlotResonancesMesh`). In the latter case, a preliminary interpolation step (using `scipy.interpolate .RectBivariateSpline`) is first performed, in order to known the value of the three resonances on the cells centers of the computational mesh. In figure 5 we show the resonances location for the computational grid (and thus the equilibrium) reported in figure 4. For the computation of the upper-hybrid resonance, we assume the following simple expression for the electron density `ne = lambda x, y: np.abs(1.0e17*(1 - x ** 2) * (1 - y ** 2))`, implemented in Python as a simple $2D$ `lambda` function. From the figure, one can see that two resonances occurs within the computational domain. The first, Ω_{CE}, depends only on the magnetic field strength, while the second, Ω_{UH} also on the local plasma density.

6 Python code

```
import numpy as np
import scipy
from scipy.special import ellipk, ellipe
import matplotlib
import numpy.matlib
import matplotlib.pyplot as plt
import scipy.interpolate
matplotlib.rcParams.update({'font.size':
     18, 'font.family': 'serif', '
    mathtext.fontset': 'stix'})

class Coils:
    """
    Class to represent the coils defining
      the equilibrium of the machine
    """
    def __init__(self):
        """
        Initialise the main coil properties
        starting from a .txt file
        """
        string = input('Insert coil data: ')
        CoilParam = np.genfromtxt(string)
        Ncoils = np.shape(CoilParam)[1]
        self.Ncoils = Ncoils
        for ii in range(0, np.shape(
        CoilParam)[0]):
            self.Rc1 = CoilParam[0, :]
            self.Zc1 = CoilParam[1, :]
            self.Rc2 = CoilParam[2, :]
            self.Zc2 = CoilParam[3, :]
            self.Ic = CoilParam[4, :]
        self.Jc = Coils.ComputeCurrents(self
        )
    def ComputeCurrents(self):
        """
        Compute the coils current density,
        assuming that the coil is a square
        """
        Jc = np.zeros(self.Ncoils)
        Jc = self.Ic / ( (self.Rc2 - self.
        Rc1) * (self.Zc2 - self.Zc1) )
        return Jc


class Equilibrium:
    """
    Class to represent the Equilibrium of
      the machine as computed from the
      coils current density
    """
    def __init__(self, r1, r2, z1, z2, Nr,
      Nz, Nc = 10):
        """
        Initialise the domain where to
        compute the Equilibrium
        """
        self.__mu0 = 4 * np.pi * 1.0E-07
        self.r1 = r1
        self.r2 = r2
        self.z1 = z1
        self.z2 = z2
        self.Nr = Nr
        self.Nz = Nz
        self.Nc = Nc
        self.Coil = Coils()
        R, Z = Equilibrium.BuildGrid(self)
        self.R = R
        self.Z = Z
        self.psi = Equilibrium.ComputePsi(
        self)
        Br, Bz = Equilibrium.ComputeField(
        self)
        self.Br = Br
        self.Bz = Bz
    def BuildGrid(self):
        """
        Compute the cartesian grid for the
        computation
        """
        r1, r2, z1, z2, Nr, Nz = self.r1,
        self.r2, self.z1, self.z2, self.Nr,
        self.Nz
        r = np.linspace(r1, r2, Nr)
        z = np.linspace(z1, z2, Nz)
        R, Z = np.meshgrid(r, z)
        return R, Z
    def BuildGridCoils(self, index):
        """
        Compute a cartesian grid on each of
        the (index) coil with a predefined (
        Nc=10) number of points
        """
        Rc1, Rc2, Zc1, Zc2, Nc = self.Coil.
        Rc1, self.Coil.Rc2, self.Coil.Zc1,
    self.Coil.Zc2, self.Nc
    zc = np.linspace(Zc2[index], Zc1[
    index], Nc)
    rc = np.linspace(Rc2[index], Rc1[
    index], Nc)
    Rc, Zc = np.meshgrid(rc, zc)
    return Rc, Zc, rc, zc
  def ComputePsi(self):
    """
    Compute the poloidal flux function
    using the Green Function method
    """
    R, Z, Nr, Nz, Nc, Jc, Ncoil, mu0 =
    self.R, self.Z, self.Nr, self.Nz,
    self.Nc, self.Coil.Jc, self.Coil.
    Ncoils, self.__mu0
    psi_coil = np.zeros((Nz, Nr, Ncoil))
    for ll in range(0, Ncoil):
      G = np.zeros((Nz, Nr, Nc, Nc))
      Rc, Zc, rc, zc = Equilibrium.
    BuildGridCoils(self, ll)
      for kk in range(0, Nc):
        for mm in range(0, Nc):
          k2 = ( 4.0 * R * Rc[kk,mm] ) /
   ( (R + Rc[kk,mm]) ** 2 + (Z - Zc[kk
  ,mm]) ** 2)
          k = np.sqrt(k2)
          K = ellipk(k2)
          E = ellipe(k2)
          G[:, :, kk,mm] = Jc[ll] * ((
    mu0 / ( 2 * np.pi)) * np.sqrt( R *
    Rc[kk,mm]) / k ) * ( (2 - k2) * K -
    2 * E)
      psi_coil[:,:,ll] = np.trapz(np.
    trapz(G, zc), rc)
    psi = np.zeros((Nz, Nr))
    for ii in range(0, Ncoil):
      psi = psi + psi_coil[:, :, ii]
    return psi
  def ComputeField(self):
    """
    Function for the computation of the
    radial and axial magnetic field
    components
    """
    Nr, Nz, r2, r1, z2, z1, psi, R =
    self.Nr, self.Nz, self.r2, self.r1,
    self.z2, self.z1, self.psi, self.R
    Br = np.zeros((Nz, Nr))
    Bz = np.zeros((Nz, Nr))
    dr = (r2 - r1) / (Nr - 1)
    dz = (z2 - z1) / (Nz - 1)
    # Compute the axial magnetic field
    for ii in range(0, Nz):
      for jj in range(0, Nr):
        if jj == 0:
          Bz[ii, jj] = 2 * (psi[ii,jj +
  1] - psi[ii, jj]) / (dr*dr)
        elif jj == Nr - 1:
          Bz[ii, jj] = (psi[ii,jj] - psi
  [ii,jj - 1]) / dr
          Bz[ii, jj] = Bz[ii, jj] / R[ii
  ,jj]
        else:
          Bz[ii,jj] = (psi[ii, jj + 1] -
   psi[ii, jj - 1]) / (2 * dr)
          Bz[ii,jj] =  Bz[ii, jj] / R[ii
  ,jj]
    # Compute the radial magnetic field
    for ii in range(0, Nz):
      for jj in range(0, Nr):
        if ii == 0:
          Br[ii, jj] = (psi[ii + 1, jj]
  - psi[ii, jj]) / dz
        elif ii == Nz - 1:
          Br[ii, jj] = (psi[ii, jj] -
  psi[ii - 1, jj]) / dz
        else:
          Br[ii, jj] = (psi[ii + 1, jj]
  - psi[ii - 1, jj]) / (2 * dz)
        Br[ii,jj] = - Br[ii,jj] / R[ii,
  jj]
    return Br, Bz


class Mesh:
  """
  Class for the definition and
    construction of a field-aligned
    structured Mesh
  """
  def __init__(self, Eq, m, n, Rlim):
    self.Eq = Eq
    self.m = m
    self.n = n
    self.__Ncont = 100
    self.__LowerPsiValue = 1.0E-08
    self.__Dim = 24
    self.Rlim = Rlim
    PsiLim = Mesh.ComputeLastPsi(self)
    Mesh.PsiLim = PsiLim
    self.Mesh = Mesh.ComputeGrid(self)
def ComputeLastPsi(self):
    psi, R, Z, LowerPsiValue, Ncont,
   Rlim = self.Eq.psi, self.Eq.R, self.
   Eq.Z, self.__LowerPsiValue,  self.
   __Ncont, self.Rlim
    ContourLevels = np.linspace(
   LowerPsiValue, psi.max(), Ncont)
    Cnt = plt.contour(R, Z,psi,
   ContourLevels)
    lines = []
    for it in range(0, Ncont):
        for line in Cnt.collections[it].
   get_paths():
            lines.append(line.vertices)
        r = np.array(lines[it])[:, 0].max
   () - Rlim
        if r >= 0:
            print('Found LCFS at: ',
   ContourLevels[it - 1])
            break
    return ContourLevels[it - 1]
def InterpolatePsi(self, zP, rP,
   zQuery, ContourLevels):
    psi = self.Eq.psi
    for ii in range(0, ContourLevels.
   size):
        Cnt = np.array(plt.contour(rP, zP,
    psi, [ContourLevels[ii]]).allsegs)
        plt.ioff()
        Shape = Cnt.shape
        r = Cnt.reshape(Shape[2], 2)[:, 0]
        z = Cnt.reshape(Shape[2], 2)[:, 1]
        InterpFunc = scipy.interpolate.
   interp1d(z, r, fill_value="
   extrapolate")
        if ii == 0:
            InterpPsi = InterpFunc(zQuery).
   reshape(1, zQuery.size)
        else:
            InterpPsi = np.append(InterpPsi,
    InterpFunc(zQuery).reshape(1,
   zQuery.size), axis=0)
    return InterpPsi.T
def InterpolateField(self, rQuery,
   MESH):
    psi, Bz, R, Z, n, m = self.Eq.psi,
   self.Eq.Bz, self.Eq.R, self.Eq.Z,
   self.n, self.m
    zQuery = MESH[:, 3]
    BzC = np.zeros(n * (m + 2))
    InterpFunc = scipy.interpolate.
   RectBivariateSpline(Z[:, 0], R[0,
   :], Bz)
    for ii in range(0, zQuery.size):
        BzC[ii] = InterpFunc(zQuery[ii],
   rQuery[ii])
    return BzC
def ComputeGrid(self):
    LowerPsiValue, PsiLim, m, n, Z, R,
   z2 = self.__LowerPsiValue, self.
   PsiLim, self.m, self.n, self.Eq.Z,
   self.Eq.R, self.Eq.z2
    ContourLevels = np.linspace(np.sqrt(
   LowerPsiValue), np.sqrt(PsiLim), m)
   ** 2
    zValues = Z[:, 0]
    rValues = R[0, :]
    alpha = 1.8
    overshoot = 1.0E-05
    func = lambda x : z2 * ( ( np.tanh(-
   x)) / (np.tanh(alpha)) * (1+
   overshoot)) / (1 + overshoot)
    IndexFunc = np.linspace(-alpha,
   alpha, 2 * n + 1)
    zQuery = func(IndexFunc)
    InterpPsi = Mesh.InterpolatePsi(self
   , zValues, rValues, zQuery,
   ContourLevels)
    crxT = zQuery[0 : zQuery.size - 1 :
   2]
    crxB = zQuery[2 : zQuery.size : 2]
    cryLT = InterpPsi[0 : InterpPsi.
   shape[0] - 1: 2, 0 : InterpPsi.shape
   [1] - 1].T.reshape(1, n * (m - 1))
    cryRT = InterpPsi[0 : InterpPsi.
   shape[0] - 1: 2, 1 : InterpPsi.shape
   [1]].T.reshape(1, n * (m - 1))
    cryLB = InterpPsi[2 : InterpPsi.
   shape[0] : 2, 0 : InterpPsi.shape[1]
    - 1].T.reshape(1, n * (m - 1))
    cryRB = InterpPsi[2 : InterpPsi.
   shape[0] : 2, 1 : InterpPsi.shape
   [1]].T.reshape(1, n * (m - 1))
    cryBottomGuard = np.zeros((1, n))
    cryUpperGuard = Mesh.InterpolatePsi(
   self, zValues, rValues, zQuery, np.
   array([1.0E-12]))
    cryUpperGuardLT = cryUpperGuard[0 :
   cryUpperGuard.size - 1: 2]
    cryUpperGuardLB = cryUpperGuard[2 :
   cryUpperGuard.size : 2]
    cryLT = np.append(np.append(
   cryBottomGuard, cryUpperGuardLT),
   cryLT)
    cryLB = np.append(np.append(
   cryBottomGuard, cryUpperGuardLB),
   cryLB)
    cryRT = np.append(np.append(
   cryUpperGuardLT, cryLT[2 * n: 3 * n
   ]), cryRT)
    cryRB = np.append(np.append(
   cryUpperGuardLB, cryLB[2 * n: 3 * n
   ]), cryRB)
    cryLB = np.append(cryLB, cryRB[cryRB
   .size - n: cryRB.size])
    cryLT = np.append(cryLT, cryRT[cryRB
   .size - n: cryRB.size])
    cryRT = np.append(cryRT, cryRT[cryRB
   .size - n: cryRB.size] + 1.0E-05)
    cryRB = np.append(cryRB, cryRB[cryRB
   .size - n: cryRB.size] + 1.0E-05)
    cryC = (cryLT + cryRB) / 2
    crxC = (crxT + crxB) / 2
    MESH = np.empty((n * (m + 2), 14))
    MESH[:, 2] = cryC
    MESH[:, 3] = np.matlib.repmat(crxC,
   1, m + 2)
    MESH[:, 4] = cryLT
    MESH[:, 5] = np.matlib.repmat(crxT,
   1, m + 2)
    MESH[:, 6] = cryLB
    MESH[:, 7] = np.matlib.repmat(crxB,
   1, m + 2)
    MESH[:, 8] = cryRT
    MESH[:, 9] = np.matlib.repmat(crxT,
   1, m + 2)
    MESH[:, 10] = cryRB
    MESH[:, 11] = np.matlib.repmat(crxB,
    1, m + 2)
    MESH[:, 0] = np.matlib.repmat(np.
   array([range(0, n)]), 1, m + 2)
    MESH[:, 1] = np.matlib.repeat(np.
   array([range(0, m + 2)]), n)
    BzC = Mesh.InterpolateField(self,
   cryC, MESH)
    MESH[:, 12] = BzC.reshape(n * (m  +
   2))
    MESH[:, 13] = 0
    return MESH
  def PlotMesh(self):
    Mesh = self.Mesh
    it = 0
    Mesh = np.delete(Mesh, (1), axis =
   1)
    Mesh = np.delete(Mesh, (0), axis =
   1)
    Shape = Mesh.shape
    A = np.zeros(Shape[0] * (Shape[1] -
   2))
    for ii in range(0, Shape[0]):
      for jj in range(0, Shape[1] - 2):
        A[it] = Mesh[ii, jj]
        it = it + 1
    x = A[0 : A.size : 2]
    y = A[1 : A.size : 2]
    Points = self.n * 5
    fig, ax = plt.subplots()
    for ii in range(0, self.m + 3):
      ax.plot(x[ii * self.n * 5 + 1 :
   Points], y[ii * self.n * 5 + 1 :
   Points], 'black', linewidth = 0.5)
      Points = Points + self.n * 5
    ax.set_aspect(aspect=0.3)
    ax.set_xlabel('R [m]')
    ax.set_ylabel('Z [m]')
    plt.show()
  def WriteMesh(self):
    Mesh = self.Mesh
    Name = input('Insert Mesh name: ') +
     '.ASCII'
    np.savetxt(Name, Mesh, '%.8E')
    print('Saved Mesh in file: ' + Name)

class PlotEquilibrium:
  """
  Class to plot the Equilibrium object
  """
  def __init__(self, Eq, Ncont = 100,
   Dim = 24):
    self.Equ = Eq
    self.Ncont = Ncont
    self.Dim = Dim
```

```
def PlotPsiContour(self):
    """
    Contour plot of psi
    """
    R, Z, psi, Ncont = self.Equ.R, self.
   Equ.Z, self.Equ.psi, self.Ncont
    fig, ax = plt.subplots()
    cnt = ax.contour(R, Z, psi, Ncont,
   cmap=matplotlib.cm.RdGy, vmin=abs(
   self.Equ.psi).min(), vmax=abs(self.
   Equ.psi).max())
    cb = fig.colorbar(cnt, ax=ax)
    plt.show()
def PlotBContour(self):
    """
    Contour plot of B
    """
    R, Z, Br, Bz, Ncont = self.Equ.R,
   self.Equ.Z, self.Equ.Br, self.Equ.Bz
   , self.Ncont
    B = np.sqrt(self.Equ.Br ** 2 + self.
   Equ.Bz ** 2)
    fig, ax = plt.subplots()
    cnt = ax.contourf(R, Z, B, Ncont,
   cmap=matplotlib.cm.RdGy, vmin=abs(B)
   .min(), vmax=abs(B).max())
    ax.set_xlabel('R [m]')
    ax.set_ylabel('Z [m]')
    cb = fig.colorbar(cnt, ax=ax)
    plt.show()
def PlotPsiBContour(self):
    """
    Plot psi and B contour on the same
   figure
    """
    R, Z, Br, Bz, psi, Ncont = self.Equ.
   R, self.Equ.Z, self.Equ.Br, self.Equ
   .Bz, self.Equ.psi, self.Ncont
    B = np.sqrt(self.Equ.Br ** 2 + self.
   Equ.Bz ** 2)
    fig, ax = plt.subplots(nrows = 2,
   ncols = 1)
    cntPsi = ax[0].contour(R[0, :], Z[:,
    0], psi, Ncont, cmap=matplotlib.cm.
   RdGy,vmin=abs(psi).min(), vmax=abs(
   psi).max())
    cb = fig.colorbar(cntPsi, ax = ax
   [0])
    ax[0].set_xlabel('R [m]')
    ax[0].set_ylabel('Z [m]')
    cntB = ax[1].contourf(R, Z, B, Ncont
   , cmap=matplotlib.cm.RdGy,vmin=abs(B
   ).min(), vmax=abs(B).max())
    cb = fig.colorbar(cntB, ax = ax[1])
    ax[1].set_xlabel('R [m]')
    ax[1].set_ylabel('Z [m]')
    plt.show()

class Resonances:
    """Class for the computation of the
      various resonances in the machine"""
    def __init__(self, OmegaInj, Eq, Mesh,
     neFit):
        self.OmegaInj = OmegaInj
        self.Eq = Eq
        self.Mesh = Mesh
        self.__e = 1.6e-19
        self.__me = 9.11e-31
        self.__mi = 2 * 1.672e-27
        self.neFit = neFit
        self.OmegaCE, self.OmegaUH, self.
       OmegaLH, self.OmegaPE = Resonances.
       ComputeResonances(self)
        self.OmegaC1, self.OmegaC2 =
       Resonances.ComputeCutOff(self)
        self.ZResPosition = Resonances.
       LocateAxial(self)
    def ComputeResonances(self):
        Br, Bz, e, me, mi, neFit, Eq = self.
       Eq.Br, self.Eq.Bz, self.__e, self.
       __me, self.__mi, self.neFit, self.Eq
        B = np.sqrt(Br ** 2 + Bz ** 2)
        OmegaCE = e * B / (2 * np.pi * me)
        OmegaCI = e * B / (2 * np.pi * mi)
        OmegaPE = 8.98 * np.sqrt(neFit(Eq.R,
         Eq.Z))
        OmegaPI = 1 / (2 * np.pi) * np.sqrt(
       me / mi) * OmegaPE
        OmegaUH = np.sqrt(OmegaPE ** 2 +
       OmegaCE ** 2)
        OmegaLH = np.sqrt(OmegaCE * OmegaCI
       * (1 + OmegaCI ** 2 / OmegaPI ** 2)
       / (1 + OmegaCE ** 2 / OmegaPE ** 2))
        return OmegaCE, OmegaUH, OmegaLH,
       OmegaPE
    def ComputeCutOff(self):
        OmegaCE, OmegaPE = self.OmegaCE,
       self.OmegaPE
```

```
        OmegaC1 = 0.5 * (np.sqrt(OmegaPE **
       2 + OmegaCE ** 2) - OmegaCE) / (2 *
       np.pi)
        OmegaC2 = 0.5 * (np.sqrt(OmegaPE **
       2 + OmegaCE ** 2) + OmegaCE) / (2 *
       np.pi)
        return OmegaC1, OmegaC2
    def LocateAxial(self):
        """Method for the computation of the
         axial location of the resonance"""
        OmegaInj, Br, Bz, psi, R, Z, e, me,
       mi, OmegaCE = self.OmegaInj, self.Eq
       .Br, self.Eq.Bz, self.Eq.psi, self.
       Eq.R, self.Eq.Z, self.__e, self.__me
       , self.__mi, self.OmegaCE
        cs = plt.contour(R[0, :], Z[:, 0],
       OmegaCE, [OmegaInj])
        lines = []
        for line in cs.collections[0].
       get_paths():
            lines.append(line.vertices)
        zpos = np.empty(len(lines))
        for ii in range(0, len(lines)):
            zpos[ii] = np.array(lines[ii]) [:,
         1].min()
        return zpos
    def PlotResonances(self):
        """Method for plotting of the
       resonances"""
        OmegaInj, OmegaCE, OmegaUH, OmegaLH,
         R, Z = self.OmegaInj, self.OmegaCE,
         self.OmegaUH, self.OmegaLH, self.Eq
       .R, self.Eq.Z
        fig, ax = plt.subplots(nrows = 1,
       ncols = 3, figsize = (10, 10))
        ax[0].contour(R[0, :], Z[:, 0],
       OmegaCE, [OmegaInj])
        ax[0].set_title('$\Omega_{CE}$')
        ax[0].set_xlabel('R [m]')
        ax[0].set_ylabel('Z [m]')
        ax[1].contour(R[0, :], Z[:, 0],
       OmegaUH, [OmegaInj])
        ax[1].set_title('$\Omega_{UH}$')
        ax[1].set_xlabel('R [m]')
        ax[1].set_ylabel('Z [m]')
        ax[2].contour(R[0, :], Z[:, 0],
       OmegaLH, [OmegaInj])
        ax[2].set_title('$\Omega_{LH}$')
        ax[2].set_xlabel('R [m]')
        ax[2].set_ylabel('Z [m]')
        fig.tight_layout()
    def PlotCutOff(self):
        """Method for plotting of the
       cutoffs"""
        OmegaInj, OmegaC1, OmegaC2, R, Z=
       self.OmegaInj, self.OmegaC1, self.
       OmegaC2, self.Eq.R, self.Eq.Z
        fig, ax = plt.subplots(nrows = 1,
       ncols = 2, figsize = (10, 10))
        ax[0].contour(R[0, :], Z[:, 0],
       OmegaC1, [OmegaInj])
        ax[0].set_title('$\Omega_{C1}$')
        ax[0].set_xlabel('R [m]')
        ax[0].set_ylabel('Z [m]')
        ax[1].contour(R[0, :], Z[:, 0],
       OmegaC2, [OmegaInj])
        ax[1].set_title('$\Omega_{C2}$')
        ax[1].set_xlabel('R [m]')
        ax[1].set_ylabel('Z [m]')
        fig.tight_layout()
    def PlotResonancesMesh(self):
        """Method for plotting of the
       resonances on the physical mesh"""
        OmegaCE, OmegaUH, OmegaLH, R, Z,
       Grid, n, m, OmegaInj = self.OmegaCE,
         self.OmegaUH, self.OmegaLH, self.Eq
       .R, self.Eq.Z, self.Mesh.Mesh, self.
       Mesh.n, self.Mesh.m, self.OmegaInj
        OmegaCEInterp = scipy.interpolate.
       RectBivariateSpline(Z[:, 0], R[0,
       :], OmegaCE)
        OmegaUHInterp = scipy.interpolate.
       RectBivariateSpline(Z[:, 0], R[0,
       :], OmegaUH)
        OmegaLHInterp = scipy.interpolate.
       RectBivariateSpline(Z[:, 0], R[0,
       :], OmegaLH)
        Rc, Zc = Grid[:, 2], Grid[:, 3]
        OmegaCEGrid = np.zeros(n * (m + 2))
        OmegaUHGrid = np.zeros(n * (m + 2))
        OmegaLHGrid = np.zeros(n * (m + 2))
        for ii in range(0, Rc.size):
            OmegaCEGrid[ii] = OmegaCEInterp(Zc
       [ii], Rc[ii])
            OmegaUHGrid[ii] = OmegaUHInterp(Zc
       [ii], Rc[ii])
            OmegaLHGrid[ii] = OmegaLHInterp(Zc
       [ii], Rc[ii])
```

```
 OmegaCEGrid = OmegaCEGrid.reshape(m
+ 2, n)
 OmegaUHGrid = OmegaUHGrid.reshape(m
+ 2, n)
 OmegaLHGrid = OmegaLHGrid.reshape(m
+ 2, n)
 it = 0
 Grid = np.delete(Grid, (1), axis =
1)
 Grid = np.delete(Grid, (0), axis =
1)
 Shape = Grid.shape
 A = np.zeros(Shape[0] * (Shape[1] -
2))
 for ii in range(0, Shape[0]):
   for jj in range(0, Shape[1] - 2):
     A[it] = Grid[ii, jj]
     it = it + 1
 x = A[0 : A.size : 2]
 y = A[1 : A.size : 2]
 Points = n * 5
 Rc = Rc.reshape(m + 2, n)
 Zc = Zc.reshape(m + 2, n)
 fig, ax = plt.subplots(nrows = 1,
ncols = 3, figsize = (10, 10))
 ax[0].contour(Rc, Zc, OmegaCEGrid, [
OmegaInj])
 ax[0].set_title('$\Omega_{CE}$')
 ax[0].set_xlabel('R [m]')
 ax[0].set_ylabel('Z [m]')
 for ii in range(0, m + 3):
   ax[0].plot(x[ii * n * 5 + 1 :
Points], y[ii * n * 5 + 1 : Points],
  'black', linewidth = 0.5)
   Points = Points + n * 5
 Points = n * 5
 ax[1].contour(Rc, Zc, OmegaUHGrid, [
OmegaInj])
 ax[1].set_title('$\Omega_{UH}$')
 ax[1].set_xlabel('R [m]')
 ax[1].set_ylabel('Z [m]')
 for ii in range(0, m + 3):
   ax[1].plot(x[ii * n * 5 + 1 :
Points], y[ii * n * 5 + 1 : Points],
  'black', linewidth = 0.5)
   Points = Points + n * 5
 Points = n * 5
 ax[2].contour(Rc, Zc, OmegaLHGrid, [
OmegaInj])
 ax[2].set_title('$\Omega_{LH}$')
 ax[2].set_xlabel('R [m]')
 ax[2].set_ylabel('Z [m]')
 for ii in range(0, m + 3):
   ax[2].plot(x[ii * n * 5 + 1 :
Points], y[ii * n * 5 + 1 : Points],
  'black', linewidth = 0.5)
   Points = Points + n * 5
 Points = n * 5
 plt.show()
 fig.tight_layout()
```