Manual de uso de códigos estandarizados para FreeGS

Andrés Camacho López[1] and **Luis Vincen López Hernández**[2]

[1]Facultad de Ciencias, Universidad Nacional Autónoma de México
[2]Área Académica de Física y Matemáticas, Universidad Autónoma del Estado de Hidalgo
[1]andr3s@ciencias.unam.mx
[2]lo463591@uaeh.edu.mx

18 de Julio de 2025

Resumen

El presente manual tiene como objetivo la descripción y explicación de 3 tipos de códigos estandarizados que facilitan el estudio del problema del equilibrio de frontera libre en tokamaks, como se hizo específicamente en este manual para MAST-U y DIII-D por medio de FreeGS, la creación de estos códigos, responde a la necesidad de homologar y optimizar la obtención de datos que FreeGS puede brindar.

Introducción

FreeGS

FreeGS es un módulo de Python que calcula el equilibrio de plasma para experimentos de fusión nuclear en tokamaks, este módulo se basa en solucionar la ecuación de Grad-Shafranov con frontera libre.

Desarrollo teórico importante

Aunque la finalidad no de este manual no es desarrollar por completo el método numérico por el cual funciona este solucionador, se incluirán algunas cosas que no estan completamente descritas en [1]. Artículo en el cual basa FreeGS su funcionamiento.
En dispositivos toroidales y axisimétricos, como los tokamaks, la relación de balance de fuerza, se describe a través de una ecuación diferencial parcial de segundo órden no lineal y elíptica [2], esta ecuación es la ecuación de Grad-Shafranov

$$\Delta^* \psi(R, Z) = -\mu_0 R J_{\phi,pl}(R, Z), \tag{1}$$

debido a que esta ecuación involucra el operador elíptico $\Delta^*\psi$, cuya obtención no es trivial, pues involucra la manipulación de la función de Green,se recomienda antes revisar [3] donde se puede encontrar un desarrollo detallado de este operador, el cual no es común encontrar completo en libros de texto sobre física de plasmas, además en caso de no estar familiarizado con la aplicación de funciones de Green a problemas de electromagnetismo, tambíen es recomendable revisar el capitulo 2 de [4] donde se aborda el uso de la función de Green para estas soluciones.
Una parte importante de la ecuación de Grad-Shafranov (1) y por lo tanto del solucionador, es la densidad de corriente el plasma, el cual se puede ver en el término $J_{\phi,pl}$, este término, se puede desarrollar mas a detalle reescribiendo la ecuación de Grad-Shafranov.
Partiendo de la ecuación para un plasma confinado magnéticamente

$$\nabla p = \vec{J} \times \vec{B}, \tag{2}$$

se pueden desarrollar las componentes del termino del lado derecho, recodando la geometría de un tokamak,

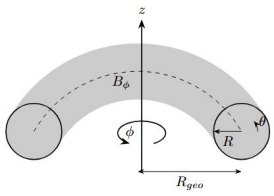


Figura 1: Diagrama de la geometría de un tokamak. ϕ hace referencia la coordenada toroidal φ y θ a la estructura poloidal p.

obteniendo como resultado

$$\begin{aligned}\vec{J}\times\vec{B} &= (J_p + J_\varphi)\times(B_p + B_\varphi)\\ &= J_p\times B_\varphi + J_\varphi \times B_p\\ &= \frac{1}{\mu_0}(\nabla\times B_\varphi)\times B_\varphi + \frac{1}{\mu_0}(\nabla\times B_p)\times B_p.\end{aligned} \tag{3}$$

Ahora si recordamos que el campo magnético para un tokamak se puede dividir como:

$$\vec{B} = (B_p) + (B_\varphi) = (\nabla\psi\times\nabla\varphi) + (F\nabla\varphi), \tag{4}$$

donde la igualdad derecha es el desarrollo de cada componente del campo magenético $\vec{B}$.

Usando las ecuaciones (3) y (4) podemos obtener las siguientes relaciónes por separado:

$$\begin{aligned}\mu_0\cdot J_p\times B_\varphi &= (\nabla F\times\nabla\varphi)\times F\nabla\varphi\\ &= -F|\nabla\varphi|^2\nabla F\\ &= -\frac{F}{R^2}\nabla F\\ &= -\frac{1}{R^2}F(\psi)\frac{dF}{d\psi}\nabla\psi,\end{aligned} \tag{5}$$

y también

$$\begin{aligned}\mu_0\cdot J_\varphi\times B_p &= (-\Delta^*\psi\nabla\varphi)\times(\nabla\psi\times\nabla\varphi)\\ &= -\frac{\Delta^*\psi}{R^2}\nabla\psi,\end{aligned} \tag{6}$$

para esta relación se nota que $\mu_0 J_\varphi\hat{e}_\varphi = \nabla\times\nabla\times(\psi\nabla\psi) = -\Delta^*\psi\nabla\varphi$ y que $B_p = \nabla\times(\psi\nabla\varphi)$.

Finalmente podemos igualar los términos de la ecuación original (1), desarrollado antes que $\nabla p = \frac{dp}{d\psi}\nabla\psi$, por lo que obtenemos:

$$\begin{aligned}\frac{dp}{d\psi}\nabla\psi &= -\frac{\Delta^*\psi}{R^2}\frac{\nabla\psi}{\mu_0} - \frac{1}{R^2}\frac{F(\psi)}{\mu_0}\frac{dF}{d\psi}\nabla\psi\\ \Rightarrow -\frac{\Delta^*\psi}{R^2} &= \mu_0\frac{dp}{d\psi} + \frac{1}{R^2}F(\psi)\frac{dF}{d\psi},\end{aligned} \tag{7}$$

finalmente llegamos a la reescritura de la ecuación de Grad-Shafranov:

$$\Delta^*\psi = -\mu_0 R^2\frac{dp}{d\psi} - F\frac{dF}{d\psi} = -\mu_0 R^2 p'(\psi) - FF'(\psi). \tag{8}$$

Es de esta ecuación que se obtiene la ecuación para la densidad de corriente:

$$J_{\phi,pl} = Rp'(\psi) + \frac{FF'(\psi)}{R\mu_0}. \tag{9}$$

Como se mencionó al inicio esta ecuación es no lineal, sin embargo, estas ecuación mostradas, solo encontramos una dependencia de ψ, por lo que la no linealidad se puede concluir revisando si ψ es una simple variable o también depende de algo más. En este caso dado que ψ corresponde a la función de flujo magnético.

$$\psi = \int_S B\cdot dA, \tag{10}$$

En la cual, se recuerda que $B = \nabla\times A$, entonces tenemos la siguiente integral:

$$\psi = \int_S (\nabla\times A)\cdot dA, \tag{11}$$

La cual por el teorema de Stokes, se convierte en:

$$\psi = \oint_{\partial S} A\cdot dl. \tag{12}$$

Para resolver esto, consideramos una superficie perpendicular al eje toroidal, es decir en el plano toroidal, y además resolviendo la integral como se menciona en [1], se obtiene:

$$\psi(R,Z) = \oint_{\partial S} A_\theta\cdot d\theta = RA_\theta(R,Z), \tag{13}$$

esto al generar otra dependencia en las ecuaciones para la densidad de corriente (9) y por lo tanto en (1) hace que la ecuaciones se vuelvan no lineales, lo que es importante pues el código basa su solución en esto.

Ya que se han hecho estas aclaraciones, en el código se introduce a la ecuación de densidad de corriente en forma canónica:

$$J_{\phi,pl} = \lambda\left[\beta_0\frac{R}{R_{\text{geo}}}j_p(\psi,\psi_a,\psi_b) + (1-\beta_0)\frac{R_{\text{geo}}}{R}j_F(\psi,\psi_a,\psi_b)\right], \tag{14}$$

sin embargo, el código simplifica esta ecuación a:

$$J_{\phi,pl} = \lambda\left[\beta_0\frac{R}{R_{\text{geo}}} + (1-\beta_0)\frac{R_{\text{geo}}}{R}\right]j_p(\psi,\psi_a,\psi_b). \tag{15}$$

Los términos j_p y j_F se pueden igualar pues j_F no altera de manera significativa el cálculo de la densidad de corriente, para poder entender como funciona cada parámetro, se recomienda revisar el código interactivo de densidad de corriente en [5]. También dado que en los resultados que obtenemos de FreeGS nos interesa obtener el perfil de densidad de corriente, además de los valores de λ y β_0, los cuales se obtienen a través de ecuaciones para I_p y β_p. Se recomienda revisar el método matemático que está programado en el archivo `jtor.py` que se puede encontrar en los archivos de módulo

de FreeGS.

Breve explicación del método numérico

Dado que la ecuación de Grad-Shafranov no es lineal, pero es posible desarrollarla, podemos resolverla de manera computacional por medio de métodos numéricos. Comenzamos definiendo una malla en la cual se resolverán ecuaciones diferenciales. Uno de los estos métodos es el uso de derivadas por diferencia numérica. En concreto este código usa derivadas centrales, es por esto que se repasará la forma en la que se llega a la ecuación de Grad-Shafranov descrita en el artículo de Young [1] que está en forma algebraica.
Primero recordando las definiciones de diferencia central para la primera y segunda derivada:

$$\begin{aligned} f'(x_0) &= \frac{f(x_0+h)-f(x_0-h)}{2h} \\ f''(x_0) &= \frac{f(x_0+h)-2f(x_0)+f(x_0-h)}{h^2} \end{aligned} \tag{16}$$

y luego usando la ecuación de Grad-Shafranov en forma diferencial:

$$\Delta^*\psi = \frac{\partial^2\psi}{\partial R^2} - \frac{1}{R}\frac{\partial\psi}{\partial R} + \frac{\partial^2\psi}{\partial Z^2}, \tag{17}$$

aplicando las diferencias centrales obtenemos:

$$\begin{aligned} \Delta^*\psi =& \frac{\psi_{j,l+1}-2\psi_{j,l}+\psi_{j,l-1}}{\Delta R^2} - \frac{\psi_{j,l+1}-\psi_{j,l-1}}{2R_l\Delta R} + \\ & \frac{\psi_{j+1,l}-2\psi_{j,l}+\psi_{j-1,l}}{\Delta Z^2}, \end{aligned} \tag{18}$$

Donde notamos que los subíndices $l \pm 1$ corresponden a la diferencia $R_0 \pm h$ y los subíndices $j \pm 1$ corresponden a la diferencia $Z_0 \pm h$, además ΔR y ΔZ toman los valores de h. Agrupando los términos correspondientes, finalmente se obtiene

$$\begin{aligned} &\frac{\psi_{l,j-1}}{\Delta Z^2} + (\frac{1}{\Delta R^2} + \frac{1}{2R_l\Delta R})\psi_{j,l-1} - 2(\frac{1}{\Delta R^2}+\frac{1}{\Delta Z^2})\psi_{j,l} + \\ &(\frac{1}{\Delta R^2} - \frac{1}{2R_l\Delta R})\psi_{j,l+1}\frac{1}{\Delta Z^2}\psi_{j+1,l} = -\mu_0 R_l J_{\phi j,l}, \end{aligned} \tag{19}$$

Ésta ecuación es la que ayudará a resolver el equilibrio, y se resuelve de forma iterativa usando iteraciones de Picard [1]. Este método, es una forma de obtener la solución a una ecuación diferencial de manera iterativa, siguiendo el el teorema de Picard – Lindelöf, que asegura la existencia de una única solución, a un problema de valores iniciales en un dominio dado $[a, b]$.

$$\frac{dy}{dt} = f(t,y); \quad y(t_0) = y_0, \tag{20}$$

Al cual se le pueden asociar las iterantes de picard en $[a, b]$ definidas por:

$$y_n(t) = y_0 + \int_{t_o}^{t} f(t, y_{n-1}(t))dt \tag{21}$$

Donde se llega a la solución por la definición debido a la convergencia puntual y convergencia uniforme de sucesión de funciones, la cual es: Sea $\{f_n\}$ una sucesión de funciones definidas en un conjunto no vacío $D \subset \mathbb{R}$, $f_n : D \longrightarrow \mathbb{R}$, y f una función $f : I \longrightarrow \mathbb{R}$ con $I \subset D$. La sucesión $\{f_n\}$ converge:
Puntualmente si:

$$\forall x \in I,\ \forall \varepsilon > 0,\ \exists N \in \mathbb{N} : n \geq N \Rightarrow |f_n(x) - f(x)| < \varepsilon. \tag{22}$$

Uniformemente si:

$$\forall \varepsilon > 0,\ \exists N \in \mathbb{N} : n \geq N \Rightarrow |f_n(x) - f(x)| < \varepsilon \forall x \in I. \tag{23}$$

Para explicar el funcionamiento del método de iteraciones de Picard, se desarrollará el siguiente ejemplo:
Supongamos la ecuación diferencial $\frac{dy}{dt} = t + y$ con la condición inicial $y(0) = 1$, es decir $y_0(t) = 1$, entonces usando la primer iteración de Picard:

$$\begin{aligned} y_1(t) &= 1 + \int_0^t (s + y_0(s))ds = 1 + \int_0^t s + 1 ds \\ y_1(t) &= 1 + \frac{s^2}{2} + s|_0^t = 1 + \frac{t^2}{2} + t, \end{aligned} \tag{24}$$

ahora haciendo la segunda iteración:

$$\begin{aligned} y_2(t) &= 1 + \int_0^t (s + y_1(s))ds = 1 + \int_0^t (s + 1 + \frac{s^2}{2} + s)ds \\ &= 1 + \frac{s^2}{2} + \frac{s^3}{6} + \frac{s^2}{2}|_0^t = 1 + t + t^2 + \frac{t^3}{6}, \end{aligned} \tag{25}$$

estas iteraciónes podrian continuar de manera infinita $n \longrightarrow \infty$ pero al definir criterios de convergencia, el código itera hasta hallar la ϵ que cumpla con algún valor menor según el criterio de convergencia dado por

$$|\psi^n - \psi^{n-1}| < \varepsilon, \tag{26}$$

al llegar a este valor se toma la solución en esa iteración final. En este ejemplo, supongamos que $\varepsilon = 0.1 + t^2$ en el intervalo $[-2.5, 2.5]$ entonces veamos que $|y_2(t) - y_1(t)| = |\frac{t^2}{2} + \frac{t^3}{6}|$ y

en el intervalo señalado se cumple que $|\frac{t^2}{2} + \frac{t^3}{6}| < \varepsilon$ por lo que podríamos dejar hasta aquí la solución.

Es importante aclarar que el ajuste de la ε puede darnos cálculos más precisos entre más pequeña sea. Debe notarse que, si se hace demasiado pequeño, puede que el solucionador no converja nunca y por lo tanto no se encuentre solución. Por el lado contrario si la ε es demasiado grande, entonces se podría encontrar solución, pero posiblemente los cálculos no sean precisos. Para mayor información sobre el método, se recomienda revisar los archivos `equilibrium.py`, `gradshafranov.py`, `multigrid.py` y `divgeo.py`.

Aspectos técnicos antes de usar los códigos

Antes de empezar a usar los código, es importante crear un entorno funcional para que no haya problemas.
Como primer paso, para facilitar la implementación de los códigos y su entorno, se recomienda instalar `Anaconda`, si bien esto puede hacerse en cualquier sistema operativo, es más fácil trabajar desde una terminal de cualquier distribución de `Linux`, de lo contrario si se está usando `Windows®` puede ejecutarse todo en el `Prompt`de `Anaconda` instalado por defecto.
Lo primero que debe hacerse es crear un ambiente virtual adecuado

```
conda create - -name <nombre_del_ambiente>
python=<version_de_python>
```

en la versión de `Python` a usar, se recomienda usar `Python3.10`. Una vez hecho esto, se abrirá el ambiente virutal, para instalar las paqueterías necesarias

```
conda activate <nombre_del_ambiente>
conda install matplotlib os pandas numpy
```

Ya que está hecho esto, es recomendable crear una carpeta únicamente para instalar y clonar los archivos de FreeGS, además de descargar los códigos ya estandrizados. Entonces se procederá a instalar y descargar FreeGS como está en las instrucciones en el `GitHub` oficial [6]. Después de esto, se debe clonar el repositorio con los códigos estandarizados [5]

```
git clone https://github.com/Andr3s3/
Freegs-pruebas.git
```

Como últimos pasos antes de ejecutar los códigos, se aclara que estos códigos fueron realizados en `Spyder` pero pueden ser usados en cualquier compilador de `Python` como `Jupyter notebook`, entonces se tendrá que instalar el compilador a elección en el ambiente virtual creado y después abrirlo desde el mismo ambiente virtual, por ejemplo:

```
conda install spyder
spyder
```

Una vez instalado y ejecutado el compilador, si es el caso como en `spyder` se debe cambiar el interprete de `Python`, por lo que se seleccionará la pestaña `Herramientas`, `preferencias` y se buscará la opción `interprete de python`, en donde se selecionará la correspondiente al ambiente virtual usado por ejemplo `/home/user/anaconda3/envs/FreeGS/bin/python`, finalmente se aplica el cambio y se acepta. Ya que se ha terminado por completo este proceso, se pueden ejecutar los códigos sin problema alguno.

Funcionamiento de los códigos

Para facilitar el uso de FreeGS para cualquier tokamak, ya sea conocido o una propuesta nueva, se realizaron códigos estandarizados que pueden extrapolarse a cualquier dispositivo. [5] A continuación se explica como funcionan tales códigos, enfocados en DIII-D y MAST-U [6].

Se recomienda leer el manual de uso de FreeGS [7]

Códigos Main

Los códigos 'Main' hacen referencia al código para la configuración de cada tokamak. Aquí se puede encontrar la configuración de arranque, así como las constricciones que se impondrán para la forma del plasma, lo cual se empleará para llevar a cabo el calculo de equilibrio que nos darán la graficación y tabulación de los resultados de interés.
A continuación, se explica cada sección de los códigos 'Main'

Librerías

En esta sección del código se importan las librerías necesarias para obtener los resultados del calculo equilibrio

```
import matplotlib
matplotlib.use("Agg")
import pandas as pd
```

```
import freegs
import os
from graf import (
    guardar_perfiles,
    calcular_densidad_corriente,
    ...
)
```

En esta sección, se debe notar que se hace uso de `matplotlib.use("Agg")` que impide el salto de ventanas separadas para la graficación de los resultados, y es útil en el momento de hacer arranque automático, en caso de querer ver como son los resultados sin entrar a la carpeta, se puede quitar esta línea.
La importación de la librerías `os` es necesaria para usar los códigos entre si, sin que existan problemas de compatibilidad. Además, permite la ejecución del programa en otros sistemas operativos.
Finalmente, es necesario importar todas las funciones que se quieran utilizar para obtener resultados desde el código graf.

Path de resultados

Esta sección se orienta a la creación de carpetas y redirección de gráficas y tablas a una sola carpeta.

```
path_resultados = os.environ.get("RESULTADOS_PATH",
                  "./Resultados_DIIID/Default")
if not os.path.exists(path_resultados):
    os.makedirs(path_resultados)
# Asegurar que la carpeta existe
# Usar `path_resultados` en lugar de rutas fijas
path_tablas = path_resultados
path_imagenes = path_resultados
```

Como se puede notar, se emplea la librería `os` para poder tener compatibilidad con los códigos posteriores de arranque automático. También, se ejemplifica la creación de las carpetas **Resultados_DIID** y **Default** como ruta de guardado.

Creación del tokamak

En esta sección se encuentra la creación del dispositivo, es decir, la ubicación de los componentes de éste, el rango en el que se va a resolver el equilibrio y el número de divisiones de la malla en la que se hará la solución.

```
tokamak = freegs.machine.DIIID()
eq = freegs.Equilibrium(tokamak=tokamak,
                        Rmin=0.1, Rmax=3.0, #Dominio radial
                        Zmin=-1.8, Zmax=1.8, #Dominio en Z
                        nx=129, ny=129, #Número de malla
                        boundary=freegs.boundary.freeBoundary
                        Hagenow)
```

En este ejemplo solo se importa el modelo de DIII-D, para el caso de MAST-U, es necesario cambiar la linea **DIIID()** por **MASTU()**. Debido a que no hay un código del diseño explícito. En esta sección se recomienda referirse a [7] y [6] para programar un tokamak, y solamente importarlo en estos códigos.

Parámetros de arranque

En esta sección se configuran los parámetros de arranque para el tokamak

```
beta = float(os.environ.get("PARAM_BETA",0.33))
I = 1533632
...
am = int(os.environ.get("PARAM_AN",1))
F = R*Bt
profiles = freegs.jtor.ConstrainBetapIp(eq,
                                        beta,
                                        I,
                                        F,
                                        am,
                                        an)
```

En este ejemplo se observan las tres formas en las que se pueden escribir los parámetros de arranque, ya sea un parámetro que se planea tener siempre fijo, o una configuración que sea compatible con un arranque automático como es el caso de los que usan `os.environ.get`. Finalmente, se configura la constricción con la que se calcula el perfil, el código FreeGS utiliza Beta o la Presión, como se explica en la documentación [6] [7].

Constricciónes, puntos de isoflujo, y solucionador

En esta sección se configura la ubicación de las constricciones que limitan la forma del plasma y la ubicación de los puntos de isoflujo que queremos que determinen la separatríz del plasma.

```
RX = 1.30
RZ = 1.0
xpoints = [(RX, -RZ) # Puntos X, donde B=0.
           (RX, RZ)]
CIX = 2.00
CIZ = 0.0
CIX1 = 1.76
```

```
...

isoflux = [(RX, -RZ, RX, RZ),...]
#Puntos de isoflujo

constrain = freegs.control.constrain
            (xpoints=xpoints,
            isoflux=isoflux)
constrain(eq)
freegs.solve(eq,
             profiles,
             constrain,
             show=True)
```

En esta sección también se llama al solucionador del equilibrio (*eq*) y se generan los perfiles.

Tabulación de corrientes de las bobinas

Esta sección genera una tabla con las corrientes de las bobinas.

```
    Coils_Current = [
    ["FC1", eq.tokamak["FC1"].current/1e+6,
    'MA'],
    ["FC2", eq.tokamak["FC2"].current/1e+6,
    'MA'],
    ... ]
headC = ['Coil', 'Current', 'MA']
Coil = pd.DataFrame(Coils_Current)
Coil.to_csv(path_tablas+'/Coil_Currents_'
            +nombre1,index=False)
```

Este resultado debe escribirse por separado, pues dependiendo del diseño del tokamak, la cantidad de bobinas es diferente, por lo tanto, no se puede incluir en el código de `graf.py`

Empleo de la rutina de graficación final

En esta sección se ejecutan todas las funciones para hacer tablas y gráficas que se importaron al inicio del código.

```
guardar_perfiles(eq, profiles, RX, RZ, CIX,
                CIZ, CIX1, CIZ1, CIX2, CIZ2,
                CIX3, CIZ3, path_tablas,
                nombre1, P,  I, R, Bt, F)
jtor, pres, Ax = calcular_densidad_corriente
                (eq, profiles)
graficar_densidad_corriente(eq, jtor, Ax,
                    path_imagenes, nombre)
graficar_factor_seguridad(eq, path_imagenes,
                            nombre)
graficar_equilibrio_final(eq, constrain,
                        path_imagenes, nombre)
graficar_energia_termica(eq, path_imagenes,
                            nombre)
...
```

Código Graf

De manera general, este código contiene una función de cálculo de densidad de corriente en los diferentes puntos del plasma y 11 funciones más encargadas de graficar y guardar datos de interés.

librerías en graf

Sobre las librerías en este código:

```
import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt
import pandas as pd
import os
```

matplotlib utilizando el modo agg, permite obtener las gráficas sin mostrarlas en nuevas ventanas pero guardándolas en formato png.
pandas es utilizado principalmente para obtener los datos en formato csv.
Por su parte, *os* facilita el manejo de las rutas para guardar los archivos que se generan.

Funciones

la primera de las funciones ***guardar_perfiles*** utiliza la solución del equilibrio, los perfiles y las variables definidas en los códigos main, su principal utilidad es guardar el perfil de arranque del reactor y parámetros que se calculan como atributos del equilibrio y los perfiles, entre estos valores se incluye la triangularidad del plasma, razón de aspecto y elongación.
calcular_densidad_corriente tiene como argumentos el equilibrio y los perfiles, a partir de esto se definen las variables *Jtor*, *pres* y *Ax*. Para manipular de mejor manera las matrices que se obtienen de los perfiles para la corriente toroidal, densidad de presión y la posición de R en el eje magnético. Éstas nuevas variables servirán como argumento

en las siguientes funciones.

```
def calcular_densidad_corriente(eq,
                                profiles):
jtor = profiles.Jtor(eq.R, eq.Z,
       eq.psi(), eq.psi_bndry)
pres = profiles.pressure(eq.psiN())
Ax = eq.magneticAxis()[0]
return jtor, pres, Ax
```

graficar_densidad_corriente al igual que los siguientes graficadores, se utiliza el *path_images* y *nombre* para guardar las imágenes en la misma carpeta y distinguir las gráficas por casos. Asigna al eje X los valores de R y al eje Y los valores de J_ϕ además, obtiene una familia de curvas correspondientes a diferentes valores de Z como se muestra en la figura 2.

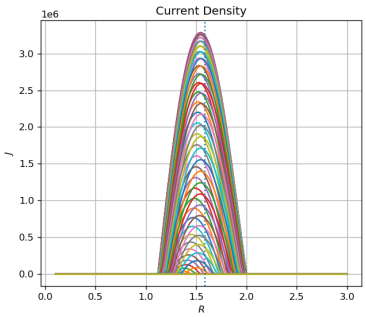


Figura 2: Familia de curvas de densidad de corriente para DIII-D

graficar_factor_seguridad utiliza la solución del equilibrio para obtener la matriz de *psi_N* y el factor de seguridad q, luego asigna el eje X y Y respectivamente para su graficación y guarda los valores de ψ_N y q en un archivo csv.

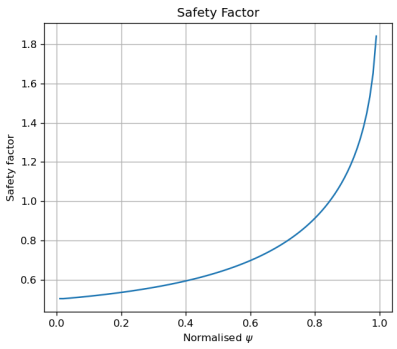


Figura 3: Factor de seguridad q típico para DIII-D

graficar_equilibrio_final utiliza como ejes los definidos en el atributo plot del equilibrio, además agrega a la imagen las constricciones definidas en los códigos main y las paredes del reactor y muestra las leyendas de cada elemento en la gráfica a un costado como se observa en las figuras 3 y 5.

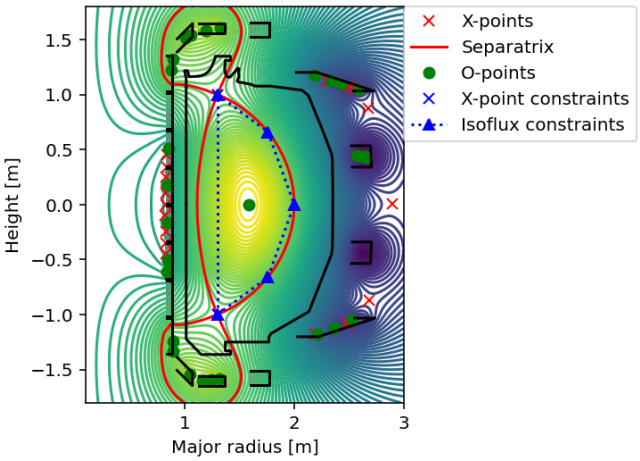


Figura 4: Gráfica de un equilibrio de DIII D

En este código se incluyen 2 funciones para graficar el equilibrio; ***graficar_equilibrio*** obtiene una visualización del equilibrio, por su parte, la función ***graficar_equilibrio_final*** implementa una pequeña pausa para optimizar el renderizado, utiliza como ejes los definidos en el atributo plot del equilibrio, además agrega a la imagen las constricciones definidas en los códigos main y las paredes del reactor y muestra las leyendas de cada elemento en la gráfica a un costado. De manera típica, es esta la gráfica que se conserva al final de la ejecución del código, pero es posible acceder a la primer visualización modificando el nombre de guardado de la imagen.

```
def graficar_equilibrio(eq, constrain,
    path_imagenes, nombre):
axis = eq.plot(show=False)
eq.tokamak.plot(axis=axis, show=False)
constrain.plot(axis=axis, show=True)
plt.savefig(os.path.join(path_imagen
            es, f'Ch6_eq_{nombre}.png'))
plt.close()
```

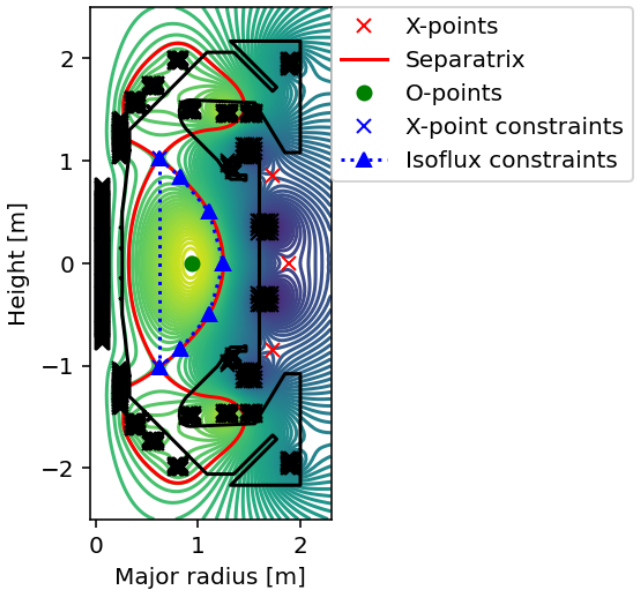


Figura 5: Gráfica de un equilibrio de MAST-U

graficar_energia_termica utiliza un if para asegurarse de obtener un valor válido antes de graficar, si se cumple la condición, asigna el tiempo al eje X y la energía térmica en Joules al eje Y.

graficar_presion_corriente, esta función obtiene los perfiles de densidad de corriente y presión como dos gráficos en una sóla imagen como se muestra en la figura 6, la obtención de estos perfiles es igual a la función ***graficar_densidad_corriente*** con la distinción de que sólo obtiene la curva correspondiente a cada perfil evaluado en Z del eje magnético.

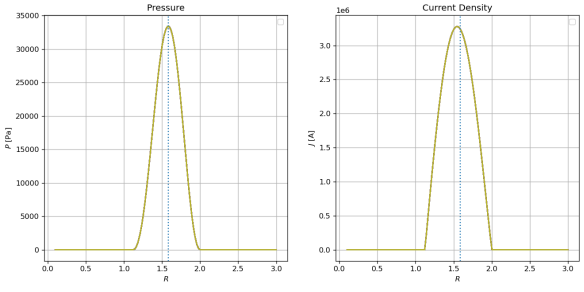


Figura 6: Gráfica perfiles de presión y corriente para DIII D

graficar_campo_magnetico Grafica los campos magnéticos radial y axial utilizando atributos del Tokamak, que se obtiene a partir de definir una variable a la cual se asigna el atributo *getmachine* del equilibrio. Por su parte el campo magnético total se obtiene de forma directa del atributo *Btot* del equilibrio, los tres gráficos se guardan en una sola imagen.

```
S1 = eq.innerOuterSeparatrix()[0]
S2 = eq.innerOuterSeparatrix()[1]
Tokamak = eq.getMachine()
```

La lineas anteriores permiten un manejo más optimo del atributo *getmachine* y nos brinda información importante que permite distinguir el comportamiento de los campos dentro y fuera de la separatriz, es importante destacar que los campos obtenidos son evaluados para la z del eje magnético. Como ejemplo, el uso de S1 y S2 se reflejan en el código de la siguiente manera:

```
plt.subplot(2, 2, 1)
plt.plot(eq.R, Tokamak.Br(eq.R, 0))
plt.axvline(Ax, ls=':')
plt.axvline(S1, ls='-.', c='red')
plt.axvline(S2, ls='-.', c='red')
plt.title(r'Campo Magnético $B_r$ Tokamak')
plt.xlabel('Radio Menor (R)')
plt.ylabel(r'$B_{r} [Teslas]$')
plt.grid()
```

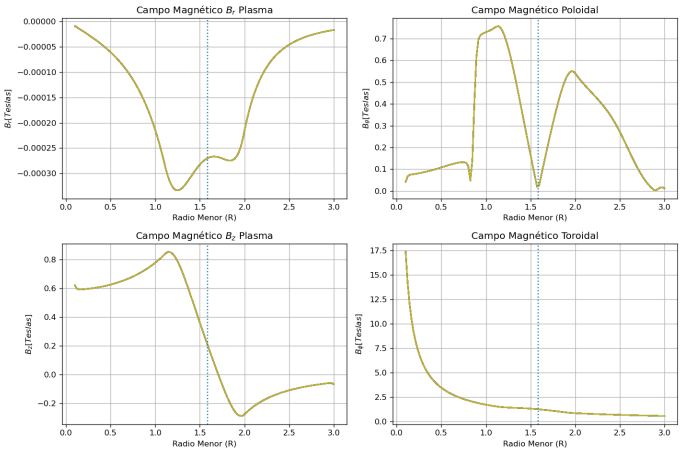


Figura 7: Gráfica de campos magnéticos radial, axial y total, linea punteada azul representa la ubicación del eje magnético en R y corresponde al equilibrio de DIII-D mostrado en la figura 4.

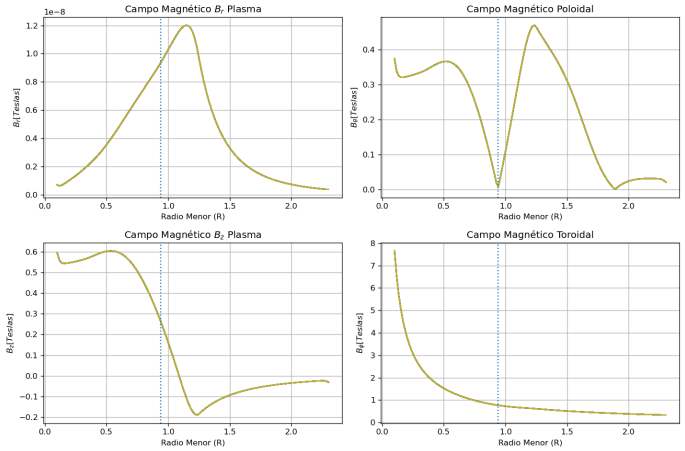


Figura 8: Gráfica de campos magnéticos radial, axial y total, linea punteada azul representa la ubicación del eje magnético en R y corresponde al equilibrio de MAST-U mostrado en la figura 5.

Análogamente ***graficar_campo_magnetico_total*** se enfoca en los campos magnéticos poloidal, toroidal, radial del plasma y axial del plasma.

graficar_separatrix grafica el conjunto de puntos que se obtienen del equilibrio en el atributo separatrix, además, muestra el valor de la triangularidad en el gráfico como se muestra en la figura 9.

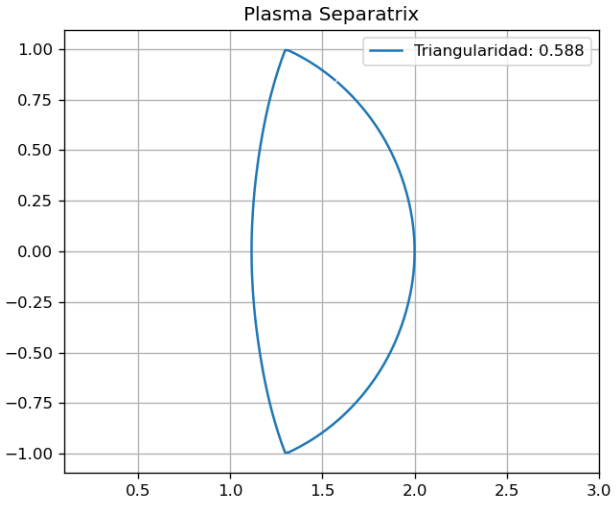


Figura 9: Gráfica de separatriz de DIII D

Por último, ***graficar_mapa_presion*** y ***graficar_mapa_corriente*** funcionan igual. Evalúan la presión y corriente en diferentes valores de ψ para construir un mapeo de la densidad de corriente y presión.

Códigos de arranque automático

Estos códigos se hicieron con la finalidad de poder obtener una serie de datos a través de un rango de valores dados, en alguno de los parámetros de arranque.

```
import subprocess
import os
import numpy as np  #Para manejar floats

# Definir el rango de valores de I con `numpy.arange()`
I_values = np.arange(100000, 1000001, 50000)
# Desde 10,000 hasta 1,000,000 con pasos de 50,000

# Carpeta base de resultados
path_resultados = "./Resultados_variando I"
if not os.path.exists(path_resultados):
    os.makedirs(path_resultados)

# Iterar sobre los valores de I y crear las subcarpetas
for I in I_values:
    carpeta_resultados = f"{path_resultados}/I_{I:.1f}"
    #Usamos formato `.1f` para nombres con decimales

    # Crear la subcarpeta antes de ejecutar `DIID.py`
    if not os.path.exists(carpeta_resultados):
        os.makedirs(carpeta_resultados)

    print(f"Ejecutando simulación para I = {I:.1f} A...")

    os.environ["RESULTADOS_PATH"] = carpeta_resultados
    os.environ["INTENSIDAD_I"] = str(I)
    #Pasar `I` para que `DIID.py` use el valor correcto

    # Imprimir ruta para verificar antes de ejecutar `DIID.py`
    print(f"RUTA ASIGNADA A RESULTADOS_PATH:{os.environ
                                    ['RESULTADOS_PATH']}")
    print(f"INTENSIDAD ASIGNADA A I: {os.environ
                                    ['INTENSIDAD_I']}")

    # Ejecutar `Main` sin modificarlo
    resultado = subprocess.run(
                        ["python", "/home/.../Main_DIIID"],
    capture_output=True, text=True)
    print(resultado.stdout)
```

En este ejemplo se muestra el código para variar la intensidad de corriente I, en este ejemplo se debe notar que sirve para parámetros que funcionan con punto decimal, en caso de que se vaya a variar un parámetro como entero, entonces cambiar las las partes necesarias del código para esto.

También se puede notar que se crean automáticamente las subcarpetas relacionadas a cada nuevo parámetro en donde se ejecutará el código `Main` y se guardarán los datos.

También será necesario asegurarse que las líneas `os.environ["INTENSIDAD_I"] = str(I)` vayan de acuerdo con la línea

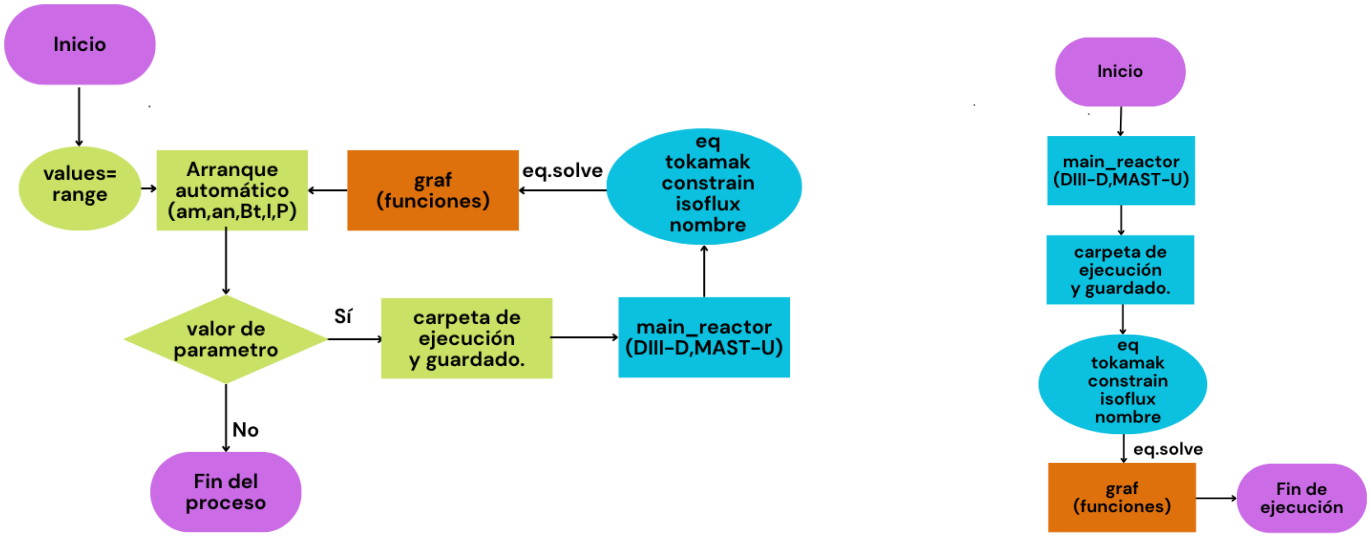
`I = float(os.environ.get("INTENSIDAD_I", 1533632))`

del código `Main`. Por último, solamente se debe llamar códi-

go `Main` con la ruta exacta en la que está almacenado, para que se ejecute en cada valor nuevo del parámetro de forma iterada como se muestra en el código.
Al final del proceso, se podrán ver en cada carpeta clasificados los resultados de cada simulación hecha.

Se recomienda que dependiendo el programa en el que se ejecute el código, (por ejemplo `spyder`) se borren las variables antes de ejecutar cualquier código de arranque automático, pues se pueden quedar guardadas en el caché y afectar los resultados que se vayan a obtener.

Diagrama de flujo



(a) Diagrama de flujo para ejecución automática.

(b) Diagrama de flujo una sóla ejecución.

Figura 10: Diagramas de flujo para ejecución de códigos.

La figura 10, muestra el flujo de ejecución de los programas, en el primer caso, es necesario establecer el perfil de arranque del primer caso de equilibrio desde main y luego ejecutar desde el código de arranque automático. Se generará un bucle de ejecución hasta que se deje de obtener un parámetro de arranque.
En el segundo caso, la ejecución es directa desde main.

Referencias

[1] Young Mu Jeon. «Development of a free-boundary tokamak equilibrium solver for advanced study of tokamak equilibria». en. En: *Journal of the Korean Physical Society* 67.5 (2015), págs. 843-853. DOI: 10.3938/jkps.67.843. URL: http://dx.doi.org/10.3938/jkps.67.843.

[2] Marco Ariola y Alfredo Pironti. *Magnetic Control of Tokamak Plasmas*. Springer, oct. de 2010. ISBN: 1849967830. URL: http://books.google.com/books?id=I9PGcQAACAAJ&dq=isbn:1849967830&hl=&source=gbs_api.

[3] Julio Herrera Velázquez. «Green function for the Grad-Shafranov operator». En: *Revista Mexicana de Física E* 19.1 Jan-Jun (ene. de 2022). DOI: 10.31349/revmexfise.19.010211. URL: https://doi.org/10.31349/revmexfise.19.010211.

[4] John David Jackson. *Classical Electrodynamics*. John Wiley y Sons, ago. de 1998.

[5] Andrés Camacho López y Uriel Yafté Sánchez Almaguer. *GitHub - Andr3s3/Freegs-pruebas*. 2025. URL: https://github.com/Andr3s3/Freegs-pruebas.git.

[6] Freegs-Plasma. *GitHub - freegs-plasma/freegs: Free boundary Grad-Shafranov solver*. URL: https://github.com/freegs-plasma/freegs.

[7] Ben Dudson. *FreeGS 0.8.3.DeV13+Gdd8701e Documentation*. 2023. URL: https://freegs.readthedocs.io/en/latest/.