Documentación de Scripts de Ploteo

Simulación Híbrida PIC 1D

Código RRK-FFT de Viñas (1993)

Scripts documentados:

`plotenergy_beam.py` – Lectura de datos Fortran
`plot_kinetic2.jl` – Lectura de datos Julia

Reproducción de la Figura 8 de
Winske & Leroy (1984)
"Diffuse Ions Produced by Electromagnetic Ion Beam Instabilities"
J. Geophys. Res., Vol. 89, No. A5, pp. 2673-2688

December 19, 2025

Contents

1 Introducción

Este documento proporciona una explicación detallada, línea por línea, de los scripts de visualización utilizados para analizar los resultados de la simulación híbrida PIC 1D. Los scripts leen el archivo binario `energy.d12` generado por la simulación y producen gráficos que reproducen la Figura 8 del paper seminal de Winske & Leroy (1984).

1.1 Archivos Documentados

Script	Lenguaje	Lee datos de
`plotenergy_beam.py`	Python 3	Código Fortran
`plot_kinetic2.jl`	Julia	Código Julia

Table 1: Scripts de ploteo y sus fuentes de datos

1.2 Estructura del Archivo `energy.d12`

El archivo `energy.d12` contiene:

1. Un **header** con parámetros de la simulación
2. Múltiples **registros** con datos de energía en cada paso de tiempo

Estructura del Header

El header contiene $N_{\text{header}} = 18 + 8 \times N_{\text{sp}}$ valores de punto flotante, donde N_{sp} es el número de especies de iones.
Para $N_{\text{sp}} = 2$: $N_{\text{header}} = 18 + 16 = 34$ valores.

Part I

Script Python para Datos Fortran

2 Encabezado y Propósito

2.1 Shebang

```
#!/usr/bin/env python3
```

Propósito: Esta línea es el *shebang* que indica al sistema operativo Unix/Linux que este script debe ejecutarse con Python 3. Permite ejecutar el script directamente como:

```
$ ./plotenergy_beam.py
```

en lugar de:

```
$ python3 plotenergy_beam.py
```

2.2 Docstring

```
"""
================================================================================
    Script para reproducir la Figura 8 de Winske & Leroy (1984)
    Lee datos del codigo Fortran hyb1d_rrkfft.f90

    DETECTA AUTOMATICAMENTE el formato del archivo energy.d12

    Genera 3 figuras:
    1. Figura 8 (4 paneles, escala log en Wf)
    2. Figura 8 (4 paneles, escala lineal)
    3. Energias globales del sistema (1 panel)
================================================================================
"""
```

Propósito: El docstring documenta el propósito general del script. Es una buena práctica de programación que ayuda a otros (y a ti mismo en el futuro) a entender qué hace el código.

3 Importaciones

3.1 NumPy

```
import numpy as np
```

Propósito: NumPy es la biblioteca fundamental para cálculo numérico en Python. Se usa para:

- Crear y manipular arrays de datos
- Operaciones matemáticas vectorizadas (más rápidas que loops)
- Leer datos binarios con `np.frombuffer()`

3.2 Matplotlib

```
import matplotlib.pyplot as plt
```

Propósito: Matplotlib es la biblioteca estándar de visualización en Python. El módulo `pyplot` proporciona una interfaz similar a MATLAB para crear gráficos.

3.3 Path

```
from pathlib import Path
```

Propósito: `Path` proporciona una forma orientada a objetos de manejar rutas de archivos. Se usa para:

- Verificar si un archivo existe: `Path(filename).exists()`
- Obtener el tamaño del archivo: `Path(filename).stat().st_size`

3.4 Struct

```
import struct
```

Crítico para Archivos Fortran

El módulo `struct` es esencial para interpretar bytes como datos empaquetados. Fortran escribe archivos binarios con "record markers" (4 bytes) antes y después de cada registro. La función `struct.unpack('i', ...)` convierte 4 bytes a un entero de 32 bits.

4 Constante Global

```
IHPARM = 18
```

Propósito: Define el número de parámetros globales en el header del archivo binario.

¿Por qué 18?

El código Fortran original define:

```
INTEGER, PARAMETER :: ihparm=18, nheadr=ihparm+8*nsp
```

Los primeros 18 valores del header son parámetros globales de la simulación (`dt`, `dx`, `nx`, etc.). Después vienen 8 parámetros por cada especie de ion.

5 Función `read_header_fortran()`

5.1 Lectura del Record Marker

```
def read_header_fortran(filename, nsp_guess=2):
    with open(filename, 'rb') as f:
        rec_marker = struct.unpack('i', f.read(4))[0]
```

Propósito: Abre el archivo en modo binario (`'rb'`) y lee los primeros 4 bytes.

Formato Binario de Fortran

Fortran escribe archivos con "record markers":

`[4 bytes: tamaño] [DATOS] [4 bytes: tamaño]`

Antes de cada bloque de datos, escribe un entero de 4 bytes indicando el tamaño del bloque en bytes.
Ejemplo: Si el header tiene 34 floats de 4 bytes cada uno, el record marker será $34 \times 4 = 136$.

Explicación del código:

- `struct.unpack('i', ...)`: El formato `'i'` significa "signed integer" de 4 bytes
- `f.read(4)`: Lee exactamente 4 bytes del archivo
- `[0]`: Extrae el valor de la tupla retornada por `unpack`

5.2 Detección Automática del Formato

```
        for nsp in [2, 1]:
            nheadr = IHPARM + 8 * nsp
            for float_size in [4, 8]:  # float32 o float64
                expected_size = nheadr * float_size
                if rec_marker == expected_size:
                    print(f"Header detectado: nsp={nsp}, float{float_size*8}")
```

Propósito: Detecta automáticamente:

1. Cuántas especies de iones hay ($N_{sp} = 1$ o 2)
2. Si los floats son de 4 bytes (`REAL` de Fortran) u 8 bytes (`DOUBLE PRECISION`)

¿Por qué es necesaria la detección automática?

El código Fortran puede compilarse con diferentes opciones:

- Con `-fdefault-real-8`: Los `REAL` son de 8 bytes
- Sin esa opción: Los `REAL` son de 4 bytes

Además, diferentes versiones del código usan diferentes números de especies.

5.3 Lectura del Header

```
                    dtype = np.float32 if float_size == 4 else np.float64
                    header = np.frombuffer(f.read(nheadr * float_size), dtype=
    dtype)
```

Propósito:

- `np.frombuffer()`: Convierte bytes crudos a un array de NumPy
- `dtype`: Especifica cómo interpretar los bytes (`float32` o `float64`)

5.4 Verificación del Marker Final

```
                    rec_end = struct.unpack('i', f.read(4))[0]
```

Propósito: Fortran escribe el mismo tamaño al final del registro. Esto sirve como verificación de integridad de los datos.

6 Función parse_header()

```
def parse_header(header, nsp):
    params = {
        'dt': float(header[0]),
        'dx': float(header[1]),
        'nx': int(header[2]),
        'itmax': int(header[3]),
        'lfld': int(header[4]),
        'nsp': int(header[5]),
        'np': int(header[6]),
        'npg': int(header[7]),
        'betaen': float(header[8]),
        'wpiwci': float(header[9]),
        'bf0': [float(header[10]), float(header[11]), float(header[12])],
        'ifield': int(header[13]),
        'iparticles': int(header[14]),
        'ienergy': int(header[15]),
        'ifilter': int(header[16]),
        'itrestart': int(header[17]),
    }
```

Propósito: Convierte el array numérico del header en un diccionario con nombres descriptivos.

¿Por qué los índices específicos?

El código Fortran escribe el header en este orden exacto:

```
header(1)=dt
header(2)=dx
header(3)=float(nx)
header(4)=float(itmax)
...
```

Los índices en Python empiezan en 0, por lo que `header[0]` corresponde a `header(1)` de Fortran.

6.1 Parámetros por Especie

```
    for is_ in range(nsp):
        indh = IHPARM + 8 * is_
        params[f'rdn_{is_+1}'] = float(header[indh])
        params[f'vdr_{is_+1}'] = float(header[indh + 1])
        params[f'betain_{is_+1}'] = float(header[indh + 2])
        params[f'anis_{is_+1}'] = float(header[indh + 3])
        params[f'qi_{is_+1}'] = float(header[indh + 4])
        params[f'ai_{is_+1}'] = float(header[indh + 5])
        params[f'nions_{is_+1}'] = int(header[indh + 6])
        params[f'gg_{is_+1}'] = float(header[indh + 7])
```

Offset	Parámetro	Descripción
+0	rdn	Densidad relativa de la especie
+1	vdr	Velocidad de drift (en V_A)
+2	betain	Beta de la especie ($8\pi nT/B_0^2$)
+3	anis	Anisotropía de temperatura ($T_\perp/T_\parallel$)
+4	qi	Carga (en unidades de e)
+5	ai	Masa (en unidades de m_p)
+6	nions	Número de partículas de esta especie
+7	gg	Factor de normalización

Table 2: Parámetros por especie en el header

Propósito: Lee los 8 parámetros de cada especie de ion.

¿Por qué `indh = IHPARM + 8 * is_`? Cada especie tiene 8 parámetros, almacenados consecutivamente después de los parámetros globales.

7 Función `detect_energy_format()`

```
def detect_energy_format(filename, header_bytes, nsp, float_size):
    with open(filename, 'rb') as f:
        f.seek(header_bytes)
        rec_marker = struct.unpack('i', f.read(4))[0]
```

Propósito: Salta el header y lee el marker del primer registro de datos para determinar su formato.

¿Por qué `f.seek(header_bytes)`? Posiciona el cursor de lectura justo después del header (incluyendo sus record markers).

7.1 Formatos Posibles

```
    formats = {
        'original': 2 + 3*nsp + 6,          # tim, tpal, tper, umx, enrgys
        'scalar_wk': 2 + 3*nsp + 2 + 6,     # + wkpal_total, wkper_total
        'array_wk': 2 + 5*nsp + 6,          # + wkpal(nsp), wkper(nsp)
    }
```

Propósito: Define los tres formatos posibles de registros de energía.

Formato	Floats	Contenido
original	$2 + 3N_{\text{sp}} + 6$	tim, tpal, tper, umx, enrgys
scalar_wk	$2 + 3N_{\text{sp}} + 2 + 6$	+ wkpal_total, wkper_total
array_wk	$2 + 5N_{\text{sp}} + 6$	+ wkpal(nsp), wkper(nsp)

Table 3: Formatos de registro de energía

¿Por qué hay diferentes formatos?

El código Fortran ha evolucionado. Versiones más nuevas guardan más información (energías cinéticas por especie). El script detecta automáticamente qué versión generó

el archivo.

8 Función `read_energy_file_fortran()`

8.1 Cálculo del Número de Registros

```
    file_size = Path(filename).stat().st_size
    data_bytes = file_size - header_bytes
    n_records = data_bytes // record_total
```

Propósito: Calcula cuántos registros de datos hay en el archivo.

Fórmula:

$$N_{\text{registros}} = \frac{\text{tamaño\_archivo} - \text{tamaño\_header}}{\text{tamaño\_por\_registro}} \tag{1}$$

donde `record_total = 4 + record_bytes + 4` (incluyendo los markers).

8.2 Loop de Lectura

```
    for i in range(n_records):
        marker_data = f.read(4)
        if len(marker_data) < 4:
            break
        rec_start = struct.unpack('i', marker_data)[0]

        if rec_start != record_bytes:
            print(f"ADVERTENCIA registro {i}: marker={rec_start}, esperado={
    record_bytes}")
            break
```

Propósito:

- Lee el marker de inicio de cada registro
- Verifica que coincida con el tamaño esperado
- Si no coincide, algo está mal y se detiene la lectura

8.3 Parsing de Datos

```
            idx = 0
            step[i] = int(data_raw[idx])
            time[i] = data_raw[idx + 1]
            idx += 2

            tpal[i, :] = data_raw[idx:idx + nsp]
            idx += nsp

            tper[i, :] = data_raw[idx:idx + nsp]
            idx += nsp
```

Propósito: Extrae cada campo del registro usando un índice que avanza.

¿Por qué usar `idx`? Los datos están empaquetados secuencialmente. El índice rastrea la posición actual en el array de datos crudos.

9 Función plot_figure8_fortran()

9.1 Extracción de Energías según Formato

```
    if energy_format == 'array_wk':
        wkpal = data['wkpal']
        wkper = data['wkper']

        if nsp >= 2:
            Wm_par = wkpal[:, 0]
            Wm_perp = 2.0 * wkper[:, 0]  # Corregir factor 2
            Wb_par = wkpal[:, 1]
            Wb_perp = 2.0 * wkper[:, 1]
```

Propósito: Extrae las energías cinéticas por especie.

Corrección del Factor 2

¿Por qué `2.0 * wkper`?
El código de simulación calcula:

$$\texttt{wkper} = \frac{1}{2}\sum_n (v_{y,n}^2 + v_{z,n}^2) \cdot \text{fac2} \tag{2}$$

Pero la energía cinética perpendicular **física** es:

$$K_\perp = \frac{1}{2}(K_y + K_z) = \frac{1}{2} \cdot \frac{1}{2} m \sum_n (v_{y,n}^2 + v_{z,n}^2) \tag{3}$$

El factor $\frac{1}{2}$ adicional en la definición de `wkper` significa que debemos multiplicar por 2 para obtener la energía física correcta.

9.2 Normalización

```
    xlen = params['xlen']
    W0_mag = xlen / 2.0
    W0_kin = Wm[0] + Wb[0]
    W0 = W0_kin if W0_kin > 0 else 1.0
```

Propósito: Define las energías de referencia para normalizar.

Energías de Normalización

¿Por qué `W0_mag = xlen / 2.0`?
En unidades normalizadas donde $B_0 = 1$, la energía magnética del campo de fondo es:

$$W_{B,0} = \frac{B_0^2 L}{8\pi} = \frac{L}{2} \quad \text{(en unidades normalizadas)} \tag{4}$$

¿Por qué usar `W0_kin`?
Normalizar por la energía cinética inicial hace que las energías sean $\mathcal{O}(1)$, facilitando la comparación con el paper de Winske & Leroy.

9.3 Creación de la Figura

```
    fig, axes = plt.subplots(2, 2, figsize=(12, 10))
```

Propósito: Crea una figura con 4 paneles (2 filas × 2 columnas).

- `figsize=(12, 10)`: Tamaño en pulgadas (ancho, alto)

9.4 Panel (a): Energías del Beam

```
    ax = axes[0, 0]
    if nsp >= 2 and np.any(Wb_par_norm > 0):
        ax.plot(t, Wb_par_norm, color=colors['par'], lw=2, label=r'$W_{b\
parallel}$')
        ax.plot(t, Wb_perp_norm, color=colors['perp'], lw=2, label=r'$W_{b\perp}
$')
```

Propósito: Grafica las energías paralela y perpendicular del beam.

¿Por qué `r'$W_{b\parallel}$'`?

- `r'...'`: "Raw string" (no interpreta \)
- `$...$`: Activa el modo matemático de LaTeX
- `\parallel`: Produce el símbolo ∥

9.5 Panel (d): Escala Logarítmica

```
    ax.semilogy(t, Wf_norm_mag, color=colors['Wf'], lw=2, label=r'$W_f$')
```

Propósito: `semilogy` usa escala logarítmica en el eje Y.

¿Por qué escala logarítmica?

La energía de fluctuaciones magnéticas crece **exponencialmente** durante la fase lineal de la inestabilidad:

$$W_f(t) \propto e^{2\gamma t} \tag{5}$$

donde γ es la tasa de crecimiento. En escala log, esto aparece como una línea recta.

10 Función `plot_global_energies()`

```
    W_B = enrgys[:, 0]      # Energia magnetica de fluctuaciones
    W_E = enrgys[:, 1]      # Energia electrica
    K_par = enrgys[:, 2]    # Energia cinetica paralela total
    K_perp = enrgys[:, 3]   # Energia cinetica perpendicular total
    K_tot = enrgys[:, 4]    # Energia cinetica total
    E_tot = enrgys[:, 5]    # Energia total del sistema
```

Propósito: Extrae cada componente del array `enrgys`.

Índice	Variable Fortran	Descripción
0	bfldenrgy	Energía magnética $\delta B^2/8\pi$
1	efldenrgy	Energía eléctrica $E^2/8\pi$
2	wkpalenrgy	Energía cinética paralela total
3	wkperenrgy	Energía cinética perpendicular total
4	wktotenrgy	Energía cinética total
5	totenrgy	Energía total del sistema

Table 4: Contenido del array enrgys

10.1 Diagnóstico de Conservación

```
    dE = (E_tot[-1] - E_tot[0]) / E_tot[0] * 100
    print(f"  Delta E/E_0 = {dE:.4f} %")
```

Propósito: Calcula el error en la conservación de energía como porcentaje.

Verificación de Conservación

Un error grande ($> 10\%$) indica problemas numéricos en la simulación:

- Paso de tiempo demasiado grande
- Número insuficiente de partículas
- Problemas en el esquema numérico

11 Función main()

```
def main():
    import sys

    energy_file = sys.argv[1] if len(sys.argv) > 1 else "energy.d12"
```

Propósito: Permite especificar el archivo como argumento de línea de comandos.

Uso:

```
$ python3 plotenergy_beam.py              # usa "energy.d12"
$ python3 plotenergy_beam.py otro.d12     # usa "otro.d12"
```

```
if __name__ == "__main__":
    main()
```

Propósito: Solo ejecuta main() si el script se ejecuta directamente, no si se importa como módulo en otro script.

Part II

Script Julia para Datos Julia

12 Encabezado

```
#!/usr/bin/env julia
```

Propósito: Shebang para ejecutar directamente como `./plot_kinetic2.jl`.

```
#=
================================================================================
    Script para reproducir la Figura 8 de Winske & Leroy (1984)
    ...
================================================================================
=#
```

Propósito: Comentario multi-línea en Julia (sintaxis `#= ... =#`).

13 Importaciones

```
using Printf
```

Propósito: Proporciona `@sprintf` para formateo de strings estilo C.

```
using CairoMakie
```

Propósito: Biblioteca de visualización de alta calidad.

Ventajas de CairoMakie sobre Plots.jl

- Mejor renderizado vectorial (PDF, SVG)
- Más control sobre la apariencia
- Mejor soporte para LaTeX en etiquetas
- Figuras de calidad publicación

14 Constantes

```
const nsp = 2
const ihparm = 18
const nheadr = ihparm + 8 * nsp
```

Propósito: Define las mismas constantes que el código de simulación.

¿Por qué `const`?

En Julia, declarar constantes permite:

- Optimizaciones del compilador (el valor es conocido en tiempo de compilación)

- Evitar reasignaciones accidentales
- Mejor inferencia de tipos

Diferencia con Python: Julia requiere que `nsp` sea conocido en tiempo de compilación para dimensionar arrays estáticos.

15 Función `read_header()`

```
function read_header(filename::String)
    header = zeros(Float64, nheadr)
    open(filename, "r") do f
        read!(f, header)
    end
```

Propósito: Lee el header del archivo binario de Julia.

Diferencia CRÍTICA con Fortran

Julia escribe datos binarios **SIN record markers**:

Fortran: `[4 bytes] [DATOS] [4 bytes]`
Julia: `[DATOS]`

Por eso usamos `read!(f, header)` directamente, sin parsear bytes extra.

```
    params = Dict{String, Any}(
        "dt" => header[1],
        "dx" => header[2],
        ...
    )
```

Propósito: Crea un diccionario con tipos mixtos (`String` → `Any`).

¿Por qué `Any`? Porque los valores pueden ser `Float64`, `Int`, o arrays.

15.1 Loop sobre Especies

```
    for is in 1:nsp
        indh = ihparm + 8 * (is - 1)
        params["rdn_$is"] = header[indh + 1]
        params["vdr_$is"] = header[indh + 2]
        ...
    end
```

Propósito: Lee los parámetros de cada especie.

¿Por qué `(is - 1)`? En Julia los índices empiezan en 1, pero el offset sigue la convención de 0-based del código original de Fortran.

¿Por qué `"rdn_$is"`? Interpolación de strings en Julia usa `$`.

16 Función read_energy_file()

16.1 Cálculo del Tamaño de Registro

```
record_size = 2 + nsp_file + nsp_file + nsp_file + 6 + nsp_file + nsp_file
```

Propósito: Calcula cuántos Float64 hay por registro.

Campo	Tamaño	Descripción
tim	2	[paso, tiempo]
tpal	N_{sp}	Temperatura paralela por especie
tper	N_{sp}	Temperatura perpendicular por especie
umx	N_{sp}	Velocidad media por especie
enrgys	6	Energías globales
wkpal_sp	N_{sp}	Energía cinética $\parallel$ por especie
wkper_sp	N_{sp}	Energía cinética $\perp$ por especie
Total	$2 + 5N_{sp} + 6$	

Table 5: Estructura de un registro de energía en Julia

Para $N_{sp} = 2$: $2 + 10 + 6 = 18$ floats por registro.

16.2 Loop de Lectura

```
open(filename, "r") do f
    seek(f, header_bytes)

    for i in 1:n_records
        tim = zeros(Float64, 2)
        read!(f, tim)
        step[i] = Int(tim[1])
        time[i] = tim[2]
```

Propósito:

- seek(f, header_bytes): Salta el header
- read!(f, tim): Lee directamente en el array pre-asignado
- Int(tim[1]): Convierte el paso de tiempo a entero

Ventaja de read!

read! lee directamente en un array existente sin crear arrays temporales. Esto es más eficiente en memoria y velocidad que crear nuevos arrays en cada iteración.

17 Función plot_figure8_corrected()

17.1 Corrección del Factor 2

```
    Wm_par = wkpal_sp[:, 1]            # W_m|| (correcta)
    Wm_perp = 2.0 .* wkper_sp[:, 1]    # W_m_perp (CORREGIDA: factor 2)
    Wb_par = wkpal_sp[:, 2]            # W_b|| (correcta)
    Wb_perp = 2.0 .* wkper_sp[:, 2]    # W_b_perp (CORREGIDA: factor 2)
```

Propósito: Aplica la corrección del factor 2 a las energías perpendiculares.

¿Por qué `.*` (con punto)? En Julia, el punto indica operación "broadcast" (elemento por elemento). Sin el punto, `2.0 * array` también funciona para escalares, pero es buena práctica ser explícito.

17.2 Normalización

```
    xlen = params["xlen"]
    W0_mag = xlen / 2.0
    W0_kin = Wm[1] + Wb[1]
    W0 = W0_kin
```

Propósito: Igual que en Python, define las energías de referencia.

¿Por qué `Wm[1]` y no `Wm[0]`? Julia usa índices 1-based.

17.3 Parámetros de Winske

```
    f = params["rdn_2"]  # fraccion del beam
    V = params["vdr_2"]  # drift del beam en V_A
    F = f * V^2 / 2      # parametro F de Winske
```

Propósito: Calcula los parámetros adimensionales usados en el paper de Winske & Leroy (1984).

Parámetros de Winske

- f = fracción de densidad del beam respecto al total
- V = velocidad de drift del beam en unidades de V_A
- $F = fV^2/2$ = parámetro de energía del beam

Estos parámetros determinan el régimen de la inestabilidad (resonante vs. no-resonante).

17.4 Predicción Teórica

```
    dB_B0_pred = sqrt(f) * V
    println("  (delta B/B_0)_max = sqrt(f) * V = $(dB_B0_pred)")
```

Propósito: Calcula la predicción teórica de Winske (Ecuación 16 del paper).

Predicción de Saturación

La amplitud máxima de las fluctuaciones magnéticas escala como:

$$\left(\frac{\delta B}{B_0}\right)_{\max} \approx \sqrt{f} \cdot V \tag{6}$$

Esta es una predicción del análisis cuasi-lineal de la inestabilidad.

17.5 Creación de la Figura con CairoMakie

```
    fig = Figure(size = (1000, 900), fontsize = 14)
```

Propósito: Crea una figura con tamaño especificado en píxeles.

Diferencia con Matplotlib: CairoMakie usa píxeles, Matplotlib usa pulgadas.

17.6 Definición de Colores

```
    color_par = :blue
    color_perp = :red
    color_total_m = :green
    color_total_b = :orange
    color_Wf = :purple
```

Propósito: Define colores como símbolos de Julia (`:blue`).

¿Por qué símbolos? Son más eficientes que strings y es el estilo idiomático de Julia.

17.7 Creación de Ejes

```
    ax_a = Axis(fig[1, 1],
        xlabel = "Omega_i t",
        ylabel = "W / W_0",
        title = "(a) Beam Energies"
    )
```

Propósito: Crea un eje en la posición [fila 1, columna 1] de la figura.

Sintaxis de CairoMakie: `fig[i, j]` especifica la posición en el grid.

17.8 Graficación

```
    lines!(ax_a, t, Wb_par_norm, color = color_par, linewidth = 2, label = "W_b
    ||")
```

Propósito: Añade una línea al eje.

Convención de ! en Julia

En Julia, la convención es que funciones que **modifican** su argumento terminan en `!`.
Aquí `lines!` modifica `ax_a` añadiendo una línea.
Funciones sin `!` típicamente retornan un nuevo objeto sin modificar el original.

17.9 Leyenda

```
    axislegend(ax_a, position = :rt)
```

Propósito: Añade una leyenda al eje.

`:rt` = "right-top" (esquina superior derecha).

17.10 Energía Total

```
    Wtot_norm = (Wm .+ Wb .+ Wf) ./ W0
    lines!(ax_c, t, Wtot_norm, color = :black, linewidth = 1.5,
           linestyle = :dash, label = "W_tot")
```

Propósito: Calcula y grafica la energía total (para verificar conservación).

¿Por qué `.+` y `./`? Operaciones elemento por elemento (broadcasting).

17.11 Título Global

```
    Label(fig[0, :],
        text = "Energy Histories - Winske & Leroy (1984) Figure 8 (Corrected)",
        fontsize = 18, font = :bold
    )
```

Propósito: Añade un título encima de todos los paneles.

`fig[0, :]`: Fila 0 (encima de la fila 1), todas las columnas.

17.12 Guardar Figura

```
    save(output_file, fig, px_per_unit = 2)
```

Propósito: Guarda la figura como PNG.

`px_per_unit = 2`: Duplica la resolución (figura de 2000×1800 píxeles reales).

18 Función `total_energy_plots()`

```
    wbtot = enrgys[:,1]
    wetot = enrgys[:,2]
    kpal = enrgys[:,3]
    kper = enrgys[:,4]
    wktot = enrgys[:,5]
    etot = enrgys[:,6]
```

Propósito: Extrae las energías globales del array `enrgys`.

Índice Julia	Variable	Descripción
1	`bfldenrgy`	Energía magnética de fluctuaciones
2	`efldenrgy`	Energía eléctrica
3	`wkpalenrgy`	Energía cinética paralela total
4	`wkperenrgy`	Energía cinética perpendicular total
5	`wktotenrgy`	Energía cinética total
6	`totenrgy`	Energía total del sistema

Table 6: Contenido del array `enrgys` en Julia

19 Función main() y Punto de Entrada

```
function main()
    energy_file = "energy.d12"

    if !isfile(energy_file)
        println("ERROR: No se encontro el archivo '$energy_file'")
        return
    end
```

Propósito: Verifica que el archivo existe antes de intentar leerlo.

```
if abspath(PROGRAM_FILE) == @__FILE__
    if length(ARGS) >= 1
        energy_file = ARGS[1]
```

Propósito:

- `PROGRAM_FILE`: El archivo que se está ejecutando
- `@__FILE__`: El archivo donde está este código
- Solo ejecuta si el script se ejecuta directamente (no si se incluye desde otro archivo)

`ARGS`: Array de argumentos de línea de comandos.

Part III

Comparación entre Scripts

20 Diferencias en Formato de Archivo Binario

Aspecto	Fortran	Julia
Record markers	Sí (4 bytes antes y después)	No
Tipo de float	`REAL` (4B) o `REAL*8` (8B)	`Float64` (8B)
Detección	Automática por tamaño de marker	Fijo (conocido a priori)

Table 7: Diferencias en formato de archivo binario

Formato Fortran:

```
[4B: size] [HEADER: 34×4B] [4B: size] [4B: size] [RECORD 1] [4B: size] ...
```

Formato Julia:

```
[HEADER: 34×8B] [RECORD 1] [RECORD 2] ...
```

Figure 1: Comparación visual de formatos binarios

21 Diferencias en Manejo de Índices

Aspecto	Python/Fortran	Julia
Primer índice	0	1
Acceso a especie 1	`array[:, 0]`	`array[:, 1]`
Último elemento	`array[-1]`	`array[end]`
Rango	`array[0:3]` (excluye 3)	`array[1:3]` (incluye 3)

Table 8: Diferencias en indexación

22 Diferencias en Bibliotecas de Graficación

23 Complejidad de Lectura

23.1 Script Python (para Fortran)

El script de Python es **más complejo** porque debe:

1. Detectar automáticamente el formato (número de especies, tipo de float)
2. Manejar los record markers de Fortran

Operación	Matplotlib (Python)	CairoMakie (Julia)
Crear figura	`fig, axes = plt.subplots(2,2)`	`fig = Figure()`
Crear eje	(implícito en subplots)	`Axis(fig[1,1], ...)`
Graficar línea	`ax.plot(x, y)`	`lines!(ax, x, y)`
Escala log Y	`ax.semilogy()`	`yscale = log10`
Leyenda	`ax.legend()`	`axislegend(ax)`
Guardar	`plt.savefig()`	`save()`

Table 9: Equivalencias entre bibliotecas de graficación

3. Soportar múltiples formatos de registro de energía
4. Verificar integridad de datos

23.2 Script Julia

El script de Julia es **más simple** porque:

1. El formato es conocido y fijo
2. No hay record markers
3. Lectura directa con `read!()`
4. Un solo formato de registro

24 Resumen de Flujo de Ejecución

24.1 Python (`plotenergy_beam.py`)

```
main()
  |
  +---> read_energy_file_fortran()
  |       +---> read_header_fortran()
  |       |       +---> parse_header()
  |       +---> detect_energy_format()
  |       +---> [loop de lectura de registros]
  |
  +---> plot_figure8_fortran()
  |       +---> Extracción de energías según formato
  |       +---> Normalización
  |       +---> Diagnóstico (print)
  |       +---> Creación de figura 2x2
  |
  +---> plot_global_energies()
          +---> Creación de figura 1 panel
```

Figure 2: Flujo de ejecución del script Python

24.2 Julia (plot_kinetic2.jl)

```
main()
  |
  +---> read_energy_file()
  |       +---> read_header()
  |       +---> [loop de lectura de registros]
  |
  +---> plot_figure8_corrected()
  |       +---> Corrección factor 2
  |       +---> Normalización
  |       +---> Diagnóstico + parámetros de Winske
  |       +---> Creación de figura 2x2
  |
  +---> plot_figure8_log()
  |       +---> Versión con escala logarítmica
  |
  +---> total_energy_plots()
          +---> Energías globales
```

Figure 3: Flujo de ejecución del script Julia

A Referencia Rápida de Comandos

A.1 Ejecución de Scripts

```
# Python
python3 plotenergy_beam.py                      # Usa energy.d12 por defecto
python3 plotenergy_beam.py otro_archivo.d12     # Especifica archivo

# Julia
julia plot_kinetic2.jl                          # Usa energy.d12 por defecto
julia plot_kinetic2.jl otro_archivo.d12         # Especifica archivo
```

A.2 Instalación de Dependencias

```
# Python
pip install numpy matplotlib

# Julia (en el REPL)
using Pkg
Pkg.add("CairoMakie")
Pkg.add("Printf")
```

B Estructura del Header

Índice	Nombre	Tipo	Descripción
1	`dt`	Float	Paso de tiempo
2	`dx`	Float	Tamaño de celda
3	`nx`	Int	Número de celdas
4	`itmax`	Int	Número máximo de iteraciones
5	`lfld`	Int	Subpasos para campos
6	`nsp`	Int	Número de especies
7	`np`	Int	Número total de partículas
8	`npg`	Int	Partículas por celda
9	`betaen`	Float	Beta de electrones
10	`wpiwci`	Float	Ratio ω_{pi}/Ω_{ci}
11-13	`bf0`	Float[3]	Campo magnético de fondo
14	`ifield`	Int	Intervalo salida campos
15	`iparticles`	Int	Intervalo salida partículas
16	`ienergy`	Int	Intervalo salida energía
17	`ifilter`	Int	Intervalo de filtrado
18	`itrestart`	Int	Flag de reinicio
	Por cada especie (8 valores):		
+1	`rdn`	Float	Densidad relativa
+2	`vdr`	Float	Velocidad de drift
+3	`betain`	Float	Beta de la especie
+4	`anis`	Float	Anisotropía $T_\perp/T_\parallel$
+5	`qi`	Float	Carga (en e)
+6	`ai`	Float	Masa (en m_p)

Índice	Nombre	Tipo	Descripción
+7	`nions`	Int	Número de partículas
+8	`gg`	Float	Factor de normalización

Table 10: Estructura completa del header