Documentación `NLSE-solver`

Lucas Gutierrez

Septiembre 11, 2023

Este código resuelve de forma numérica la ecuación de Schrödinger no lineal usando el Interaction Picture. Este consiste en tomar el siguiente cambio de variable

$$B(\omega) = e^{-\hat{D}(\omega)z} A(\omega), \tag{1}$$

donde $A(\omega)$ es la transformada de la envolvente compleja del pulso, y $\hat{D}(\omega)$ es la transformada del operador de dispersión $D(T) = \sum_{m=2}^{\infty} \frac{\beta^{(m)}}{m!} (i\partial_T)^m$. De esta manera, la ecuación diferencial a resolver será

$$\frac{dB(\omega)}{dz} = e^{-\hat{D}(\omega)z} \hat{N}(B(\omega)), \tag{2}$$

con $\hat{N}(\omega)$ el operador no-lineal expresado en frecuencia. Para resolver esta ecuación se utiliza la función solve_ivp de `scipy`. Dependiendo el tipo de ecuación que se utilice (NLSE, GNLSE, pcNLSE, pcGNLSE) solo se cambia la forma del operador no-lineal cuando se define la ecuación diferencial a resolver.

Para simular se utilizan los siguientes códigos

- `commonfunc.py`: Contiene las funciones comunes utilizadas por los otros códigos. Entre ellas, se define correctamente a las transformadas y se tienen funciones para simular pulsos super-gaussianos y solitones. Contiene las clases `Fibra` y `Sim`, en las cuales se guardan los parámetros del sistema (parámetros físicos en `Fibra`, y parámetros de simulación en `Sim`)
- `plotter.py`: Contiene funciones para graficar las soluciones. La función `plotinst` plotea el pulso y espectro al inicio y al final de la propagación. `plotcmap` plotea un mapa de colores con la evolución del pulso y el espectro.
- `solvegnlse.py`: Utiliza `solve_ivp` para el siguiente operador no lineal

  $$\hat{N}(T) = i\left(\gamma(\omega_0) + i\gamma_1 \frac{\partial}{\partial T}\right)\left(A(T)\int_0^{\infty} R(t')|A(T-t')|^2 dt'\right)$$

  La función principal es `SolveNLS`, la cual toma como parámetros a un objeto `Sim`, `Fibra` y el pulso inicial. A su vez tiene un parámetro `raman = True/False`. Si `False`, resuelve usando la aproximación de pulso ancho (*Agrawal pág. 39*). A esta función se le puede pasar un valor `zlocs = int`, en el que se indica en cuantos puntos entre 0 y L se quiere la solución. Si no se le indica este valor, el código elige las posiciones automáticamente.
- `solvepcnlse.py`: Utiliza `solve_ivp` para el siguiente operador no lineal

  $$\hat{N}(\omega) = i\bar{\gamma}(\omega)\mathcal{F}\left(C^* G^2\right) + i\bar{\gamma}^*(\omega)\mathcal{F}\left(C^2 G^*\right),$$

  con $\bar{\gamma}(\omega) = \frac{1}{2}\left(\gamma(\omega) \times (\omega+\omega_0)^3\right)^{1/4}$, $G_\omega = \left(\gamma(\omega)/(\omega+\omega_0)\right)^{1/4} A_\omega$, $C_\omega = [(\gamma(\omega)/(\omega+\omega_0))^{1/4}]^* A_\omega$. La función `Solve_pcNLSE` toma los mismos parámetros que `SolveNLS`.
- `solvepcGNLSE.py`: Utiliza `solve_ivp` para el siguiente operador no lineal

  $$\hat{N}(\omega) = i\bar{\gamma}(\omega)\mathcal{F}\left(C^* G^2\right) + i\bar{\gamma}^*(\omega)\mathcal{F}\left(C^2 G^*\right) + if_R\bar{\gamma}^*(\omega)\mathcal{F}\left(B\int_0^{\infty} h_R(\tau)|B(t-\tau)|^2 d\tau - B|B|^2\right).$$

  La función principal `Solve_pcGNLSE` utiliza los mismos parámetros que `Solve_NLS`.

Al utilizar las funciones `SolveNLS`, `Solve_pcNLSE` y `Solve_pcGNLSE`, luego de resolver el código devuelve un vector `zlocs` con las posiciones donde se obtuvo la solución, y las matrices $\mathtt{A_W}$ y $\mathtt{A_T}$, las cuales contienen la solución para toda posición en `zlocs`.

A la función `solve_ivp` se le puede determinar la tolerancia absoluta y relativa al buscar la solución. Estos son los parámetros `rtol` y `atol` que se encuentran definidos en la funciones principales. A bajas tolerancias el código es más rápido, pero mas impreciso.

Como usar

Los parámetros necesarios para simular se introducen en los objetos definidos por las siguientes clases

Clases

```
class Sim:
    def __init__(self, puntos, Tmax):
        self.puntos = puntos           #Número de puntos sobre el cual tomar el tiempo
        self.Tmax   = Tmax
        self.paso_t = 2.0*Tmax/puntos
        self.tiempo = np.arange(-puntos/2,puntos/2)*self.paso_t
        self.dW     = np.pi/Tmax
        self.freq   = fftshift( np.pi * np.arange(-puntos/2,puntos/2) / Tmax )/(2*np.pi)


class Fibra:
    def __init__(self,L,beta2, beta3, gamma, gamma1, alpha, lambda0, TR=3e-3, fR=0.18, betas=0):
        self.L  = L         #Longitud de la fibra
        self.beta2 = beta2  #beta_2 de la fibra, para calcular GVD
        self.beta3 = beta3  #beta_3 de la fibra, para calcular TOD
        self.betas = betas  #Vector con los coeficientes beta, si != 0, sobrescribe a los otros
        self.gamma = gamma  #gamma de la fibra, para calcular SPM
        self.gamma1= gamma1 #gamma1 de la fibra, para self-steepening
        self.alpha = alpha  #alpha de la fibra, atenuación
        self.TR    = TR     #TR de la fibra, para calcular Raman
        self.fR    = fR     #fR de la fibra, para calcular Raman (y self-steepening)
        self.lambda0 = lambda0 #Longitud de onda central
        self.omega0  = 2*np.pi* 299792458 * (1e9)/(1e12) /lambda0 #Frecuencia (angular) central
        if self.beta3 != 0:
            self.w_zdw = -self.beta2/self.beta3 + self.omega0
            self.zdw   = 2*np.pi* 299792458 * (1e9)/(1e12) / self.w_zdw
        else:
            self.w_zdw = None
            self.zdw   = None
        if self.gamma1 != 0:
            self.w_znw = -self.gamma/self.gamma1 + self.omega0
            self.znw   = 2*np.pi* 299792458 * (1e9)/(1e12) /self.w_znw
        else:
            self.w_znw = None
            self.znw   = None
```

Las unidades utilizadas son las siguientes

- [Distancia] = m
- [Potencia] = W
- [Tiempo] = ps
- [Frecuencia] = THz

De esta manera, las unidades en las cuales se expresan los parámetros del sistema son

- $[\beta_2] = \frac{\text{ps}^2}{\text{m}}$
- $[\beta_3] = \frac{\text{ps}^3}{\text{m}}$
- $[\alpha] = \frac{1}{\text{m}}$
- $[\gamma_0] = \frac{1}{\text{Wm}}$
- $[L] = m$
- $[\tau_1], [\tau_2], [T_R] = ps$

A continuación se encuentra un ejemplo particular de como utilizar el código.

Ejemplo GNLSE

```
import numpy as np
from solvers.solvegnlse import SolveNLS
from common.commonfunc import SuperGauss, Soliton, Sim, Fibra, Pot, fftshift, FT, IFT
from common.plotter import plotinst, plotcmap, plt


#Parametros para la simulación
N = int(2**14)  #puntos
Tmax = 70       #ps

c = 299792458 * (1e9)/(1e12)      #Vel. luz: nm/ps

lambda0 = 1550                    #Longitud de onda central: nm
omega0  = 2*np.pi*c/lambda0

#Parametros para la fibra
L     = 400                       #Lfib:   m
b2    = -20e-3                    #Beta2:  ps^2/m
b3    = 0                         #Beta3:  ps^3/m
gam   = 1.4e-3                    #Gamma:  1/Wm
gam1  = gam/omega0*0              #Gamma1: 0
alph  = 0                         #alpha:  1/m
TR    = 3e-3*0                    #TR:     fs
fR    = 0.18*1                    #fR:     adimensional (0.18)

#Parámetros pulso gaussiano:
amp    = 1                        #Amplitud:  sqrt(W), Po = amp**2
ancho  = .4                       #Ancho T0:  ps
offset = 0                        #Offset:    ps
chirp  = 0                        #Chirp:     1/m
orden  = 1                        #Orden

#Cargo objetos con los parámetros:
sim   = Sim(N, Tmax)
fibra = Fibra(L, b2, b3, gam, gam1, alph, lambda0, TR, fR)

#Calculamos el pulso inicial
#Supergaussiano:
pulso = SuperGauss(sim.tiempo, amp, ancho, offset, chirp, orden)
#Solitón:
soliton = Soliton(sim.tiempo, ancho, fibra.beta2, fibra.gamma, orden = 1)


#------Corriendo la simulación------

zlocs, AW, AT = SolveNLS(sim, fibra, soliton, raman=True, z_locs=100)


#--------------Plots----------------

plotinst(sim, fibra, AT, AW, Wlim=[-1,1], Tlim=[-7,7])
plotcmap(sim, fibra, zlocs, AT, AW, Wlim=[-1,1], Tlim=[-7,7])
```

Output

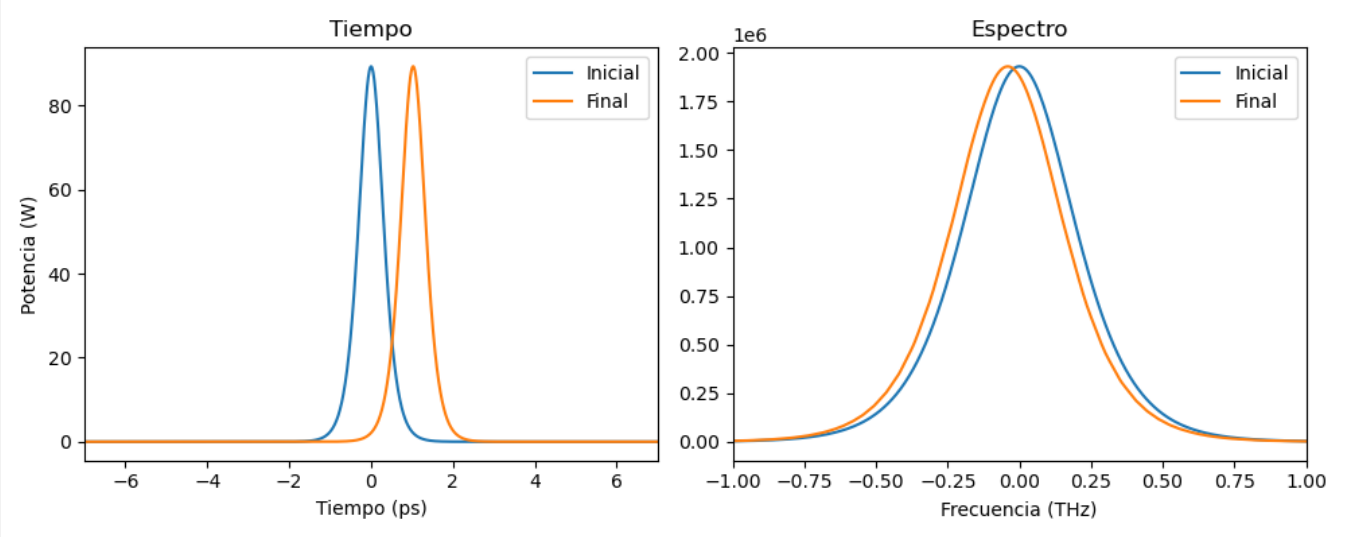

Figure 1: Pulso y espectro inicial vs final, resultado de la función `plotinst`

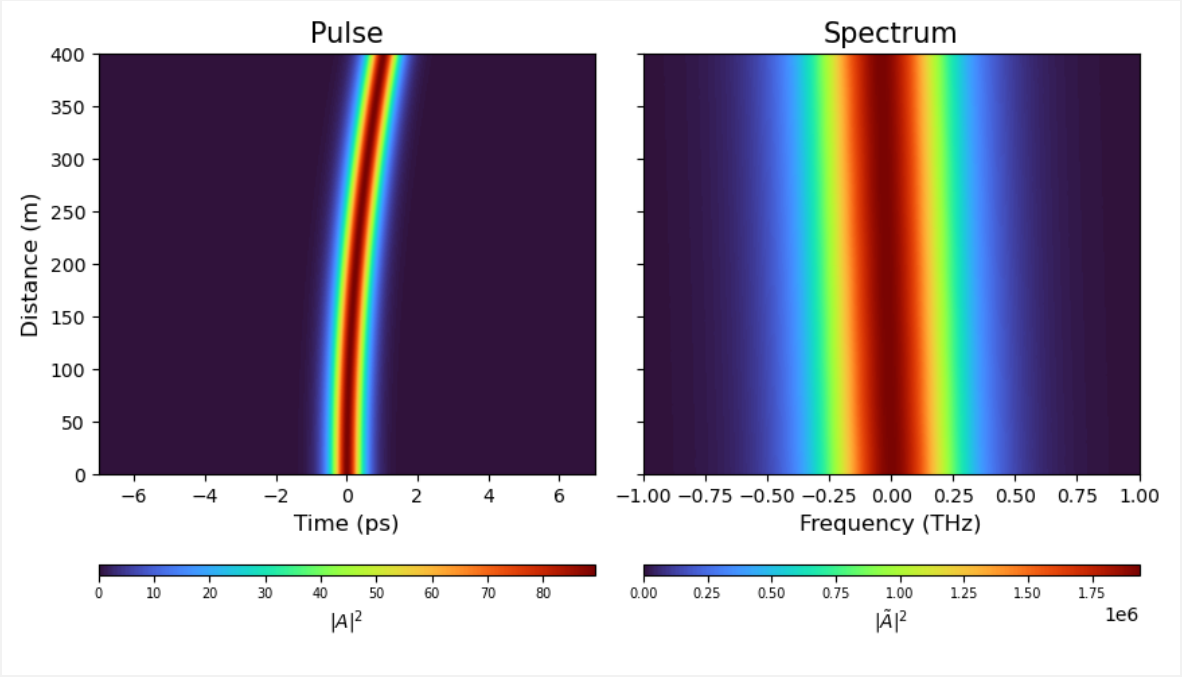

Figure 2: Mapa de colores con la evolución del pulso y espectro, resultado de la función `plotevol`

Las distintas funciones necesarias para simular estan escritas en distintos códigos .py. Para ello es necesario importarlas correctamente. El código funciona con "*imports* relativos", y para eso el directorio base de python que estemos usando es importante.

La estructura de las carpetas es

```
nlse-solver/
├── run.py
├── common/
│   ├── __init__.py
│   ├── commonfunc
│   └── plotter
└── solvers/
    ├── __init__.py
    ├── solveNLS.py
    ├── solvebarrierNLSE.py
    ├── solve_pcnlse.py
    └── solve_pcGNLSE.py
```

Para correr correctamente el código, nuestro `run.py` debe estar en la carpeta pariente `nlse-solver/`. Este archivo puede estar en una subcarpeta, siempre y cuando se le detalle a `python` que se está trabajando en `nlse-solver/`

Funciones

commonfunc.py

En primer lugar se redefinen las transformadas para seguir la convención correcta. Esto implica definir a nuestra transformada de fourier (`FT`) como la antitransformada de `scipy.fftpack`, multiplicada por un factor de normalización. Viceversa para nuestra antitransformada (`IFT`).

Transformadas

```python
#FFT siguiendo la convención -iw
def FT(pulso):
    return ifft(pulso) * len(pulso)

#IFFT siguiendo la convención -iw
def IFT(espectro):
    return fft(espectro) / len(espectro)

#Pasamos de array tiempo a array frecuencia (obsoleto)
def t_a_freq(t_o_freq):
    return fftfreq( len(t_o_freq) , d = t_o_freq[1] - t_o_freq[0])
```

Se tienen las funciones para pulsos super-gaussianos, y solitones. Se permite variar el chirp, offset y orden del super-gaussiano, así como el orden del solitón. A su vez se tiene la función `TwoPulse`, con la cual se pueden simular dos pulsos que viajen a velocidades de grupo diferentes.

Pulsos

```python
#Defino función para una super gaussiana
def SuperGauss(t,amplitud,ancho,offset,chirp,orden):
    return amplitud*np.exp(- (1+1j*chirp)/2*((t-offset)/(ancho))**(2*np.floor(orden)))*(1+0j)

#Solitón con función Sech
def Soliton(t, ancho, b2, gamma, orden):
    return orden * np.sqrt( np.abs(b2)/(gamma * ancho**2) ) * (1/ np.cosh(t / ancho) )

#Esquema para colisión de pulsos
# pulse = "s", solo signal, "p" solo pump, "sp" ambos.
def Two_Pulse(T, amp1, amp2, ancho1, ancho2, offset1, offset2, nu, pulses):
    t_1 = T/ancho1
    t_2 = T/ancho2

    np.seterr(over='ignore') #Silenciamos avisos de overflow (1/inf = 0 para estos casos)

    pump   = np.sqrt(amp1)*(1/np.cosh(t_1))
    signal = np.sqrt(amp2)*(1/np.cosh(t_2 + offset2/ancho2))*np.exp(-2j*np.pi*nu*T)

    np.seterr(over='warn') #Reactivamos avisos de overflow.

    pulse_dict = {"s": signal, "p": pump, "sp": pump + signal}
    if pulses not in pulse_dict:
        raise ValueError(f"Tipo de pulso no valido: {pulses}. Los tipos son: 's', 'p', 'sp'.")

    return pulse_dict.get(pulses)
```

Se tienen funciones auxiliares que permiten calcular la potencia, energía, número de fotones, chirp y el corrimiento en frecuencia. A su vez se encuentra la función `Adapt_Vector` que adapta al array de frecuencia y a `AW` para poder graficar en longitud de onda.

Auxiliares

```python
def Pot(pulso):
    return np.abs(pulso)**2

def Energia(t_o_freq, señal):
    return np.trapz(Pot(señal), t_o_freq)

def num_fotones(freq, espectro, w0):
    return np.sum( np.abs(espectro)**2 / (freq + w0)  )

def find_chirp(t, señal):
    fase = np.unwrap( np.angle(señal) ) #Angle busca la fase, Unwrap la extiende de [0,2pi) a
    ↪  todos los reales.
    df = np.diff( fase, prepend = fase[0] - (fase[1]  - fase[0]  ),axis=0)
    dt = np.diff( t, prepend = t[0]- (t[1] - t[0] ),axis=0 )
    chirp = -df/dt
    return chirp

def find_shift(zlocs, freq, A_wr):
    peaks = np.zeros( len(zlocs), dtype=int )
    dw    = np.copy(peaks)
    for index_z in range( len(zlocs) ):
        peaks[index_z] = np.argmax( Pot( fftshift(A_wr[index_z]) ) )
    dw = fftshift(2*np.pi*freq)[peaks]
    return dw

def Adapt_Vector(freq, omega0, Aw):
    lambda_vec = 299792458 * (1e9)/(1e12) / (freq + omega0/(2*np.pi))
    sort_indices = lambda_vec.argsort()
    lambda_vec_ordered = lambda_vec[sort_indices]

    #Esto es para identificar los distintos tipos de arrays con los que trabajamos
    if Aw.ndim == 1:
        if Aw.dtype == object:  # Aw es un array de objetos
            Alam = np.array([aw[sort_indices] for aw in Aw])
        else:  # Aw es un array 1D
            Alam = Aw[sort_indices]
    elif Aw.ndim == 2:  # Aw es un array 2D (El estandar)
        Alam = Aw[:, sort_indices]
    else:
        raise ValueError("AW debe ser un array 1D o 2D")

    return lambda_vec_ordered, Alam
```

Se tienen funciones de guardado y cargado. `saver` guarda los arrays 2d `AW` y `AT`, junto a un diccionario `metadata` con toda la información sobre la simulación. Genera dos archivos, uno llamado `filename-data.txt`, el cual guarda en formato `pickle`, y un archivo `filename-param.txt` desde el cual se pueden leer los parametros de la simulación.

La función `loader` carga los arrays, y si se elige `resim = True`, también devuelve los objetos `sim` y `fibra` ya cargados con los parámetros de la simulación.

Guardado y cargado

```
# Guardado (BAJO PRUEBA!)
def saver(AW, AT, sim:Sim, fib:Fibra, filename, other_par = None):
    # Guardando los parametros de simulación y de la fibra en un diccionario.
    metadata = {'Sim': sim.__dict__, 'Fibra': fib.__dict__} #sim.__dict__ = {'puntos'=N,
    ↪  'Tmax'=70, ...}

    # Guardando los datos en filename-data.txt con pickle para cargar después.
    with open(f"{filename}-data.txt", 'wb') as f:
        pickle.dump((AW, AT, metadata), f)

    # Guardar parametros filename-param.txt para leer directamente.
    with open(f"{filename}-param.txt", 'w') as f:
        f.write('-------------Parameters-------------\n')
        f.write(f'{datetime.datetime.now().strftime("%d-%m-%Y %H:%M:%S")}\n\n')
        for class_name, class_attrs in metadata.items():
            f.write(f'\n-{class_name}:\n\n')
            for attr, value in class_attrs.items():
                f.write(f'{attr} = {value}\n')
        if other_par:
            f.write("\n\n-Other Parameters:\n\n")
            if isinstance(other_par, str):
                f.write(f'{other_par}\n')
            else:
                for i in other_par:
                    f.write(f'{str(i)}\n')

# Cargando datos
def loader(filename, resim = None):
    with open(f"{filename}-data.txt", 'rb') as f:
        AW, AT, metadata = pickle.load(f)
    if resim:
        sim, fibra = ReSim(metadata)
        return AW, AT, sim, fibra
    else:
        return AW, AT, metadata

# Cargando metadata en las clases
# Devuelve objetos sim:Sim and fib:Fibra objects, ya cargados con los parámetros.
def ReSim(metadata):
    # Load the saved parameters in metadata to the Sim and Fibra classes, returning sim and fibra
    ↪  objects.
    sim = Sim(**metadata['Sim'])
    fibra = Fibra(**metadata['Fibra'])
    return sim, fibra
```

plotter.py

La función `plotinst` grafica el pulso/espectro inicial y final. En conjunto con dar el objeto `Sim`, A_W y A_T, se pueden aclarar los límites del gráfico con `Tlim = [tmin,tmax]` y `Wlim = [wmin,wmax]`. En conjunto se puede aclarar `save = string`, con lo cual se guardará el gráfico con el nombre elegido. Si `dB = True`, se toma al espectro en escala logarítmica. `zeros = True` permite ver los ceros de dispersión y no linealidad, `wavelength=True` devuelve el output en longitud de onda. Aclarando `end = int`, se obtiene la salida en algún otro punto.

plotinst

```
def plotinst(sim:Sim, fib:Fibra, AT, AW, Tlim=None, Wlim=None, Ylim=None, wavelength=False,
↪  zeros=None, save=None, dB=None, noshow=None, end=-1):
    #Ploteamos el pulso y espectro inicial y final:
    fig, (ax1,ax2) = plt.subplots(1,2,figsize=(10,4))
    #----pulso----
    ax1.plot( sim.tiempo, Pot(AT[0]), label="Inicial" )
    end = int(end)
    ax1.plot( sim.tiempo, Pot(AT[end]), label="Final" )
    if Tlim:
        ax1.set_xlim(Tlim)
    ax1.set_title("Tiempo")
    ax1.set_xlabel("Tiempo (ps)")
    ax1.set_ylabel("Potencia (W)")
    ax1.legend(loc="best")
    #----espectro---- Cuidado:
    if wavelength:
        lambda_vec, Alam = Adapt_Vector(sim.freq, fib.omega0, AW)
        x = lambda_vec
        y_1 = Pot(Alam[0])
        y_2 = Pot(Alam[end])
    else:
        ax2.set_title("Espectro")
        ax2.set_xlabel("Frecuencia (THz)")
        x = fftshift( sim.freq  )
        y_1 = Pot( fftshift(AW[0])  )
        y_2 = Pot( fftshift(AW[end]) )
    ax2.plot( x, y_1, label="Inicial")
    ax2.plot( x, y_2 , label="Final")
    if Wlim:
        ax2.set_xlim(Wlim)
    if Ylim:
        ax1.set_ylim(Ylim)
        ax2.set_ylim(Ylim)
    if zeros:
        freq_zdw = (fib.omega0 - fib.w_zdw)/(2*np.pi)
        freq_znw = (fib.omega0 - fib.w_znw)/(2*np.pi)
        if wavelength:
            ax2.axvline(x = fib.zdw, linestyle=":", color="gray",  label="ZDW")
            ax2.axvline(x = fib.znw, linestyle="--", color="gray", label="ZNW")
        else:
            ax2.axvline(x = freq_zdw, linestyle="--", label="ZDW")
            ax2.axvline(x = freq_znw, linestyle="--", label="ZNW")
    ax2.legend(loc="best")
    if dB:
        ax1.set_yscale("log")
        ax2.set_yscale("log")
    plt.tight_layout()
    if save:
        plt.savefig(save, dpi=800)
    if noshow:
        plt.close()
    else:
        plt.show()
```

La función `plotevol` grafica la evolución del pulso y espectro con un mapa de colores. Tiene los mismos parámetros que `plotinst`. **Código viejo, usar `plotcmap` en su lugar!**

plotevol

```
def plotevol(sim:Sim, fib:Fibra, zlocs, AT, AW, Tlim=None, Wlim=None, save=None, dB=None,
↪  wavelength=False, noshow=None, cmap="turbo"):
    #Armamos la fig y los axes
    fig, (ax1,ax2) = plt.subplots(1,2,sharey=True,figsize=(8.76,5)) #ax1: Pulso, ax2: Espectro
    #Construimos los meshgrids para graficar
    T_d, ZT = np.meshgrid(sim.tiempo, zlocs*1e-3)
    if wavelength:
        lamvec, AW = Adapt_Vector(sim.freq, fib.omega0, AW)
        W_d, ZW = np.meshgrid(lamvec, zlocs*1e-3)
    else:
        W_d, ZW = np.meshgrid(2*np.pi*sim.freq, zlocs*1e-3)
    P_T = Pot(np.stack(AT)) #Usamos np.stack para convertir los vectores AT, AW, en matrices
    P_W = Pot(np.stack(AW))
    lvl_num = 250 #Numero de colores
    #-----Plot pulso-----
    if dB:
        min_exp = -8 #Exponente mínimo, para asignar colores donde espectro = 0
        min_draw = 1.000001 * 10.**min_exp #Si valor es menor a 10**min_exp, asignamos este valor
        P_T_m = np.where(P_T < 10.**min_exp, min_draw, P_T) #Nos armamos el P_T corregido
        levels = np.logspace(np.log10(P_T_m.min()), np.log10(P_T_m.max()), lvl_num)
        #Ver que onda sin el np.log10
        surf=ax1.contourf(T_d ,ZT , P_T_m , levels=levels, cmap=cmap,norm=LogNorm())
    else:
        surf=ax1.contourf(T_d, ZT, P_T, levels = lvl_num, cmap=cmap)
    ax1.set_xlabel('Time (ps)',size=12)
    ax1.set_ylabel('Distance (km)',size=12)
    if Tlim:
        ax1.set_xlim(Tlim)
    ax1.set_title("Pulse evolution",size=15)
    #-----Plot espectro-----
    cb2 = fig.colorbar(surf, ax=ax1, label="$|A|^2$", location = "bottom", aspect=50)
    cb2.ax.tick_params(labelsize=7)
    if dB:
        min_exp = -8
        min_draw = 1.000001 * 10.**min_exp
        P_W_m = np.where(P_W < 10.**min_exp, min_draw, P_W)
        levels = np.logspace(np.log10(P_W_m.min()), np.log10(P_W_m.max()), lvl_num)
        surf=ax2.contourf(W_d ,ZW , P_W , levels=levels, cmap=cmap, norm=LogNorm() )
    else:
        surf=ax2.contourf(W_d ,ZW , P_W ,levels=lvl_num, cmap=cmap, vmin=0, vmax=.5e7)
    if wavelength:
        ax2.set_xlabel("Wavelength (nm)",size=12)
    else:
        ax2.set_xlabel('Frequency (THz)',size=12)
    if Wlim:
        ax2.set_xlim(Wlim)
    ax2.set_title("Spectrum evolution",size=15)
    cb1 = fig.colorbar(surf, ax=ax2, label="$|A|^2$", location = "bottom", aspect=50)
    cb1.ax.tick_params(labelsize=7)
    ax2.set_facecolor('black')
    ax1.tick_params(labelsize=10)
    ax2.tick_params(labelsize=10)
    plt.tight_layout()
    if save:
        plt.savefig(save, dpi=800)
    if noshow:
        plt.close()
    else:
        plt.show()
```

La función `plotcmap` grafica la evolución del pulso y el espectro con un mapa de colores. Tiene parámetros iguales a `plotinst`. El `colorbar` es interactivo, se puede arrastrar o expandir para cambiar la escala de colores dinámicamente.

plotcmap

```
#--Función requerida por "plotcmap", para cambiar los ticks a formato 10^x en escala dB--
def format_func(value, tick_number):
    return f'$10^{{{int(value/10)}}}$'
#-------------------------------------------------------------------------------------
def plotcmap(sim:Sim, fib:Fibra, zlocs, AT, AW, wavelength=False, dB=False,
             legacy=False, vlims=[], cmap="turbo", Tlim=None, Wlim=None,
             zeros=False, save=None, noshow=False, plot_type="both"):

    #Labels y tamaños
    cbar_tick_size = 7
    tick_size      = 10
    m_label_size   = 12
    M_label_size   = 15

    #Adaptamos los vectores viejos a matrices, de ser necesario
    if legacy:
        AT = np.stack(AT)
        AW = np.stack(AW)

    #Vectores de potencia, y listas "extent" para pasarle a imshow
    P_T = Pot(AT)
    textent = [sim.tiempo[0], sim.tiempo[-1], zlocs[0], zlocs[-1]]

    #Aplicamos el fftshift al espectro, o transformamos a longitud de onda de requerirlo
    if wavelength:
        #Se tiene el problema de que el vector lambda/AL no son lineales con la freq
        #Conversión freq -> lambda, corrigiendo para que sea lineal
        lamvec, AL = Adapt_Vector(sim.freq, fib.omega0, AW)
        # Armamos un vector de lambdas lineal (lamvec es no lineal, ya que viene de la freq.)
        lamvec_lin = np.linspace(lamvec.min(), lamvec.max(), len(lamvec))
        # Interpolamos los datos a nuestra nueva "grilla" lineal
        AW = np.empty_like(AL)
        for i in range(AL.shape[0]):
            interp_func = interp1d(lamvec, AL[i, :], kind='next')
            AW[i, :] = interp_func(lamvec_lin)
        P_W = Pot(AW)
        wextent = [lamvec_lin[0], lamvec_lin[-1], zlocs[0], zlocs[-1]]
    else:
        AW = fftshift(AW, axes=1)
        P_W = Pot(AW)
        wextent = [fftshift(sim.freq)[0], fftshift(sim.freq)[-1], zlocs[0], zlocs[-1]]

    #Escala dB: Normalizada al máximo de la entrada
    if dB:
        P_T = 10*np.log10(P_T) - np.max( 10*np.log10(P_T[0]) )
        P_W = 10*np.log10(P_W) - np.max( 10*np.log10(P_W[0]) )

    #Limites del colorbar
    if vlims:
        vmin_t = vlims[0]
        vmax_t = vlims[1]
        vmin_w = vlims[2]
        vmax_w = vlims[3]
    else:
        vmin_t = vmax_t = vmin_w = vmax_w = None
    #Ploteamos
    if plot_type == 'both':
        fig, (ax1,ax2) = plt.subplots(1,2,sharey=True,figsize=(8.76,5))
    elif plot_type == 'time':
```

```
        fig, ax1 = plt.subplots(1,1,figsize=(6.5,5))
    elif plot_type == 'freq':
        fig, ax2 = plt.subplots(1,1,figsize=(6.5,5))
    else:
        raise ValueError(f"Invalid plot type: {plot_type}. Valid types are: 'time', 'freq',
        ↪  'both'.")

    #fig, (ax1,ax2) = plt.subplots(1,2,sharey=True,figsize=(8.76,5))

    #---Plot en tiempo---
    if plot_type in ['time', 'both']:
        #Imshow 1
        im1 = ax1.imshow(P_T, cmap=cmap, aspect="auto", interpolation='bilinear', origin="lower",
                         extent=textent, vmin=vmin_t, vmax=vmax_t)
        ax1.tick_params(labelsize=tick_size)
        ax1.set_ylabel("Distance (m)", size=m_label_size)
        ax1.set_xlabel("Time (ps)", size=m_label_size)
        ax1.set_title("Pulse", size=M_label_size)
        #Colorbar 1: Es interactivo
        cbar1 = fig.colorbar(im1, ax=ax1, label='Normalized power (dB)' if dB else 'Power (W)',
        ↪  location="bottom", aspect=50 )
        cbar1.ax.tick_params(labelsize=cbar_tick_size)
        if Tlim:
            ax1.set_xlim(Tlim)

    #---Plot en espectro---
    if plot_type in ['freq', 'both']:
        #Imshow 2
        im2 = ax2.imshow(P_W, cmap=cmap, aspect="auto", interpolation='bilinear', origin="lower",
                         extent=wextent, vmin=vmin_w, vmax=vmax_w)
        ax2.tick_params(labelsize=tick_size)
        ax2.set_title("Spectrum", size=M_label_size)
        #Colorbar 2
        cbar2 = fig.colorbar(im2, ax=ax2, label='PSD (a.u. dB)' if dB else "PSD (a.u.)",
        ↪  location="bottom", aspect=50 )
        cbar2.ax.tick_params(labelsize=cbar_tick_size)

        if zeros:
            freq_zdw = (fib.omega0 - fib.w_zdw)/(2*np.pi)
            freq_znw = (fib.omega0 - fib.w_znw)/(2*np.pi)
            if wavelength:
                ax2.axvline(x = fib.zdw, linestyle="--", color="white",  label="ZDW")
                ax2.axvline(x = fib.znw, linestyle="--", color="crimson", label="ZNW")
            else:
                ax2.axvline(x = freq_zdw, linestyle="--", label="ZDW")
                ax2.axvline(x = freq_znw, linestyle="--", label="ZNW")
            plt.legend(loc="best")
        if Wlim:
            ax2.set_xlim(Wlim)
        if wavelength:
            ax2.set_xlabel("Wavelength (nm)", size=m_label_size)
        else:
            ax2.set_xlabel("Frequency (THz)", size=m_label_size)

    plt.tight_layout()

    if save:
        plt.savefig(save, dpi=800)
    if noshow:
        plt.close()
    else:
        plt.show()
```

La función `plotspecgram` gráfica el espectrograma a la salida de la simulación. Usa los mismos parámetros que `plotinst`, pero solo se introduce el vector `AT`.

plotspecgram

```
def plotspecgram(sim:Sim, fib:Fibra, AT, Tlim=None, Wlim=None, end=-1, save=None, dB=None,
↪  zeros=None, cmap="turbo", noshow=None, dpi=800):

    AT = AT[end,:]

    plt.figure()
    #x-axis limits
    xextent = [sim.tiempo[0], sim.tiempo[-1]]
    #Time span
    t_span = sim.tiempo[-1] - sim.tiempo[0]
    #Sampling rate
    t_sampling = len(sim.tiempo) / t_span
    #Colormap
    cmap = cmap

    #Escala del gráfico
    if dB:
        scale="dB"
    else:
        scale="linear"

    plt.specgram(AT,NFFT=700,noverlap=650,Fs=t_sampling,scale=scale,xextent=xextent,cmap=cmap)

    if zeros:
        if fib.w_zdw:
            freq_zdw = (fib.omega0 - fib.w_zdw)/(2*np.pi)
            plt.plot(xextent, [freq_zdw,freq_zdw], "--", linewidth=2, label="ZDW =
            ↪  "+str(round(fib.zdw))+" nm" )
        if fib.w_znw:
            freq_znw = (fib.omega0 - fib.w_znw)/(2*np.pi)
            plt.plot(xextent, [freq_znw,freq_znw], "--", linewidth=2, label="ZNW =
            ↪  "+str(round(fib.znw))+" nm" )
        plt.legend(loc="best")
    plt.xlabel("Time (ps)")
    plt.ylabel("Frequency (THz)")
    #plt.xticks(fontsize=15)
    #plt.yticks(fontsize=15)
    plt.tight_layout()
    if Wlim:
        plt.xlim(Tlim)
    if Tlim:
        plt.ylim(Wlim)
    if save:
        plt.savefig(save, dpi=dpi)
    if noshow:
        plt.close()
    else:
        plt.show()
```

Las funciones `plotenergia` y `plotfotones` grafican estas cantidades en función de la distancia de propagación. A la segunda función hay que aclararle el valor de `lambda0`, sino toma por defecto como longitud de onda central a 1550 nm.

Plot energía y fotones

```python
def plotenergia(sim:Sim, zlocs, AT, AW, save=None):
    energia_T = np.zeros( len(AT) )
    energia_F = np.zeros( len(AW) )
    for i in range(len(energia_F)):
        energia_T[i] = np.sum( np.abs(AT[i])**2 ) * sim.paso_t
        energia_F[i] = np.sum( np.abs(AW[i])**2 ) * sim.paso_t/len(AW[i])
    plt.plot(zlocs, energia_T)
    plt.plot(zlocs, energia_F)
    plt.title("Energía")
    plt.ylabel("E")
    plt.xlabel("z")
    if save:
        plt.savefig(save, dpi=800)
    plt.show()

def plotfotones(sim:Sim, zlocs, AW, lambda0 = 1550, save=None):

    c = 299792458 # Velocidad de la luz en m/s
    omega0 = 2 * np.pi * c / (lambda0*1e-9) #Definimos la frecuencia central, pasando lamba0 a m

    fotones = np.zeros( len(zlocs) )
    for i in range(len(fotones)):
        fotones[i] = num_fotones(sim.freq, AW[i], w0=omega0)
    plt.plot(zlocs, fotones)
    plt.title("Fotones")
    plt.ylabel("N")
    plt.xlabel("z")
    plt.show()
```

solvegnlse.py

Primero se define la función `Raman`, que dado un array de tiempo `T`, y valores τ_1, τ_2 y f_R, calcula la respuesta Raman en frecuencia $R(\omega)$. Para ello en el medio calcula la función $h_R(T)$, la normaliza y le hace una transformada.

Raman

```
def Raman(T, tau1=12.2e-3, tau2=32e-3, fR=0.18): #Agrawal pag.38: t1 = 12.2s fs, t2 = 32 fs
hR = np.zeros( len(T) )
hR[T>=0] = (tau1**2+tau2**2)/(tau1*tau2**2) * np.exp(-T[T>=0]/tau2) * np.sin(T[T>=0]/tau1)
↪  #Definimos el hR(T)
hR[T<0]  = 0

hR = fftshift(hR)  #Shifteamos para que la respuesta empiece al principio de la ventana temporal
hR = hR/np.sum(hR) #Normalizamos, tal que int(hR) = 1
hR_W = FT(hR)      #Pasamos el hR_W a frecuencia

R_W = fR * hR_W + (1-fR)*np.ones( len(T) )
return R_W
```

Se definen la ecuaciones diferenciales que determinan el problema. En el caso que se tome `raman = False`, se define la ecuación diferencial correspondiente a la aproximación de la respuesta Raman con un pulso ancho dada por `dBdz`. En el caso contrario se define la ecuación diferencial con la respuesta Raman completa dada por `dBdz_raman`.

Ambas funciones toman de entrada la coordenada `z`, la envolvente `B`, el operador de dispersión `D`, el array de frecuencia angular `w`, y valores de `gamma`. El caso con aproximación requiere del valor de `TR`, y el caso sin requiere la respuesta Raman en frecuencia `RW`, en conjunto con el valor de `gamma1` (que puede ser 0).

Ec. dif.

```
def dBdz(z,B,D,w,gamma,TR): #Todo lo que esta del lado derecho de dB/dz = ..., en el espacio de
↪  frecuencias.
    A_w = B * np.exp(D*z)
    A_t = IFT( A_w )
    return np.exp(-D*z) * FT(
        1j*gamma * Pot( A_t ) * A_t -  #i gamma |A|^2 A
    1j*gamma*TR * IFT( -1j*w* FT( Pot(A_t) ) ) * A_t  ) #i gamma TR d/dt(|A|^2) A

def dBdz_raman(z,B,D,w,gamma,gamma1,RW): #RW es la respuesta Raman en frecuencia (lo que calcula
↪  la función Raman(T))
    A_w = B * np.exp(D*z)
    A_t = IFT( A_w )
    op_nolin = 1j*(gamma + gamma1*w) * FT(A_t * IFT( RW * FT(np.abs(A_t)**2)  ) )
    return np.exp(-D*z) * op_nolin
```

La función `SolveNLS` es la que llama a `solve_ivp` para encontrar la solución a cada posición. Solo requiere como parámetros a un objeto creado con la clase `Sim` y uno con `Fibra`, en conjunto con la envolvente inicial `pulso_0` (es decir con $A(T=0)$). Se le puede aclarar `raman = True/False`, `zlocs` para elegir el número de puntos donde se quiere la solución.

Esta función dados los parámetros contenidos en `Sim` y `Fibra`, calcula el operador de dispersión y la respuesta Raman en frecuencia. Luego utiliza la función `partial`, con la cual se evalúa parcialmente a la ecuación diferencial definida por `dBdz`, introduciendo todos los parámetros y dejando como variables libres únicamente a la envolvente `B` y la coordenada `z`. Con ello, llama a `solve_ivp`, pasándole la ecuación diferencial parcialmente evaluada, los extremos de z donde se busca la solución, el pulso inicial y las tolerancias para resolver. Una vez hallada la solución $B(\omega, z)$, vuelve a la envolvente $A(T, z)$ y el espectro $A(\omega, z)$. Finalmente devuelve las posiciones donde hay solución `zlocs`, y las matrices de solución $\mathtt{A_W}$ y $\mathtt{A_T}$.

SolveNLS

```
def SolveNLS(sim: Sim, fib: Fibra, pulso_0, raman=False, z_locs=None):

    #Calculamos el espectro inicial, es lo que vamos a evolucionar.
    espectro_0 = FT(pulso_0)

    #Calculamos el operador lineal
    D_w = 1j * fib.beta2/2 * (2*np.pi*sim.freq)**2 + 1j * fib.beta3/6 * (2*np.pi*sim.freq)**3 -
    ↪  fib.alpha/2


    #Introducimos todos los parametros en la función, quedando f(z, B) = dB/dz
    if raman:
      RW = Raman(sim.tiempo, fR = fib.fR)
      f_B = partial(dBdz_raman, D = D_w, w = 2*np.pi*sim.freq, gamma = fib.gamma, gamma1 =
      ↪  fib.gamma1, RW = RW)
    else:
      f_B = partial(dBdz, D = D_w, w = 2*np.pi*sim.freq, gamma = fib.gamma, TR = fib.TR)


    #Tolerancias para integrar (Tolerancias estandar: rtol=1e-5, atol=1e-8)
    rtol = 1e-5
    atol = 1e-8

    #Usamos solve_ivp: Buscamos solución entre 0 y L
    if z_locs:
        t_eval = np.linspace(0,fib.L,z_locs)
        sol = solve_ivp(f_B, [0, fib.L], y0 = espectro_0, rtol = rtol, atol = atol,
        ↪  t_eval=t_eval)
    else:
        sol = solve_ivp(f_B, [0, fib.L], y0 = espectro_0, rtol = rtol, atol = atol)

    zlocs = sol["t"]  #Puntos de z donde tenemos B(w,z)
    ysol  = sol["y"]  #Array, en cada elemento se tiene un subarray [B(w0,z0), B(w0,z1), ...,
    ↪  B(w0,zf)]
    print(sol["message"])

    #Armar array de arrays A(w) y A(t). A_w[i] me tiene que dar el espectro en la posición
    ↪  zlocs[i].

    A_w = np.empty_like(zlocs, dtype=object)
    A_t = np.empty_like(zlocs, dtype=object)

    for j in range( len(zlocs) ):
        espectro_z = np.zeros( len(ysol) )*(1 + 0j)
        for i in range( len(ysol) ):
            espectro_z[i] = ysol[i][j]
        A_w[j] = espectro_z * np.exp( D_w * zlocs[j])
        A_t[j] = IFT( A_w[j] )

    return zlocs, A_w, A_t #Nos devuelve: zlocs = Posiciones donde calculamos la solución, A_w =
    ↪  Matriz con la evolución del espectro, A_t = Matriz con la evolución del pulso
```

solvepcnlse.py

Solve_pcNLSE

```
    def dBdz(z, B, D, w, gammaw_eff, r, r_c): #Todo lo que esta del lado derecho de dB/dz = ...,
    ↪  en el espacio de frecuencias.
    A_w = B * np.exp(D*z)  #Convertimos de B(w) -> A(w)
    B_w = r * A_w          #Cuidado con la notación! B es la envolvente, B_w
    C_w = r_c * A_w
    op_nolin = 1j * gammaw_eff * FT( np.conj(IFT(C_w)) * IFT(B_w)**2  ) + 1j *
    ↪  np.conj(gammaw_eff) * FT( IFT(C_w)**2 * np.conj(IFT(B_w))  )

    return np.exp(-D*z) * op_nolin


def Solve_pcNLSE(sim: Sim, fib: Fibra, pulso_0, z_locs=None):
    #Calculamos el espectro inicial, es lo que vamos a evolucionar.
    espectro_0 = FT(pulso_0)

    #Calculamos el operador lineal
    D_w = 1j * fib.beta2/2 * (2*np.pi*sim.freq)**2 + 1j * fib.beta3/6 * (2*np.pi*sim.freq)**3 -
    ↪  fib.alpha/2

    #Calculamos parámetros preliminares de la pcNLSE
    gammaw = fib.gamma + fib.gamma1 * (2*np.pi*sim.freq) #gamma(w), se podría extender
    r = ( gammaw / (2*np.pi*sim.freq + fib.omega0) )**(1/4)
    r_c = np.conj(r)
    gammaw_eff = (1/2)*( gammaw * (2*np.pi*sim.freq + fib.omega0)**3 )**(1/4)

    #Introducimos todos los parametros en la función, quedando f(z, B) = dB/dz
    f_B = partial(dBdz, D = D_w, w = 2*np.pi*sim.freq, gammaw_eff = gammaw_eff, r = r, r_c = r_c)

    #Tolerancias para integrar (Tolerancias estandar: rtol=1e-5, atol=1e-8)
    rtol = 1e-5
    atol = 1e-8

    #Usamos solve_ivp: Buscamos solución entre 0 y L
    if z_locs: #Si le damos número a zlocs, armamos un array con esa cantidad de puntos, donde
    ↪  guardamos la solución en dicho paso
        t_eval = np.linspace(0,fib.L,z_locs)
        sol = solve_ivp(f_B, [0, fib.L], y0 = espectro_0, rtol = rtol, atol = atol,
        ↪  t_eval=t_eval)
    else:
        sol = solve_ivp(f_B, [0, fib.L], y0 = espectro_0, rtol = rtol, atol = atol)

    zlocs = sol["t"]  #Puntos de z donde tenemos B(w,z)
    ysol  = sol["y"]  #Array, en cada elemento se tiene un subarray [B(w0,z0), B(w0,z1), ...,
    ↪  B(w0,zf)]
    print(sol["message"])

    #Armamos array de arrays A(w) y A(t). A_w[i] me tiene que dar el espectro en la posición
    ↪  zlocs[i].
    A_w = np.empty_like(zlocs, dtype=object)
    A_t = np.empty_like(zlocs, dtype=object)

    for j in range( len(zlocs) ):
        espectro_z = np.zeros( len(ysol) )*(1 + 0j)
        for i in range( len(ysol) ):
            espectro_z[i] = ysol[i][j]
        A_w[j] = espectro_z * np.exp( D_w * zlocs[j])
        A_t[j] = IFT( A_w[j] )

    return zlocs, A_w, A_t #Nos devuelve: zlocs = Posiciones donde calculamos la solución, A_w =
    ↪  Matriz con la evolución del espectro, A_t = Matriz con la evolución del puls
```

solvepcGNLSE.py

Solve_pcGNLSE

```
def dBdz(z, B, D, w, gammaw_eff, r, r_c, hR_W, fR): #Todo lo que esta del lado derecho de dB/dz
    A_w = B * np.exp(D*z)  #Convertimos de B(w) -> A(w)
    B_w = r * A_w   #Cuidado! B es la envolvente, B_w y B_t son los términos de la pcGNLSE
    C_w = r_c * A_w
    B_t = IFT(B_w)
    C_t = IFT(C_w)
    op_nolin = 1j * gammaw_eff * FT( np.conj(C_t) * B_t**2  ) + 1j * np.conj(gammaw_eff) * FT(
    ↪  C_t**2 * np.conj(B_t) ) + \
        1j * np.conj(gammaw_eff) * 2*fR * FT( B_t * IFT( hR_W  * FT(np.abs(B_t)**2) ) - B_t *
        ↪  Pot(B_t)  )
    return np.exp(-D*z) * op_nolin

def Raman(T, tau1=12.2e-3, tau2=32e-3, fR=0.18): #Agrawal pag.38: t1 = 12.2s fs, t2 = 32 fs
    hR = np.zeros( len(T) )
    hR[T>=0] = (tau1**2+tau2**2)/(tau1*tau2**2) * np.exp(-T[T>=0]/tau2) * np.sin(T[T>=0]/tau1)
    hR[T<0]  = 0
    hR = fftshift(hR)  #Shifteamos para que empiece al principio de la ventana temporal
    hR = hR/np.sum(hR) #Normalizamos, tal que int(hR) = 1
    hR_W = FT(hR)      #Pasamos el hR_W a frecuencia
    return hR_W

def Solve_pcGNLSE(sim: Sim, fib: Fibra, pulso_0, z_locs=None):
    #Calculamos el espectro inicial, es lo que vamos a evolucionar.
    espectro_0 = FT(pulso_0)
    #Calculamos el operador lineal
    D_w = 1j * fib.beta2/2 * (2*np.pi*sim.freq)**2 + 1j * fib.beta3/6 * (2*np.pi*sim.freq)**3 -
    ↪  fib.alpha/2
    #Calculamos parámetros preliminares de la pcNLSE
    gammaw = fib.gamma + fib.gamma1 * (2*np.pi*sim.freq) #gamma(w), se podría extender
    gammaw_eff = (1/2)*( gammaw * (2*np.pi*sim.freq + fib.omega0)**3 )**(1/4)
    r = ( gammaw / (2*np.pi*sim.freq + fib.omega0) )**(1/4)
    r_c = np.conj(r)
    #Calculamos la respuesta Raman
    hR_W = Raman(sim.tiempo, fR = fib.fR)
    #Introducimos todos los parametros en la función, quedando f(z, B) = dB/dz
    f_B = partial(dBdz, D = D_w, w = 2*np.pi*sim.freq, gammaw_eff = gammaw_eff, r = r, r_c = r_c,
    ↪  hR_W = hR_W, fR = fib.fR)
    #Tolerancias para integrar (Tolerancias estandar: rtol=1e-5, atol=1e-8)
    rtol = 1e-5
    atol = 1e-8
    #Usamos solve_ivp: Buscamos solución entre 0 y L
    if z_locs: #Elegimos donde
        t_eval = np.linspace(0,fib.L,z_locs)
        sol = solve_ivp(f_B, [0, fib.L], y0 = espectro_0, rtol = rtol, atol = atol,
        ↪  t_eval=t_eval)
    else: #El código elige los puntos
        sol = solve_ivp(f_B, [0, fib.L], y0 = espectro_0, rtol = rtol, atol = atol)
    zlocs = sol["t"]  #Puntos de z donde tenemos B(w,z)
    ysol  = sol["y"]  #Array, en cada elemento se tiene un subarray [B(w0,z0),...]
    print(sol["message"])
    #Armamos array de arrays A(w) y A(t).
    A_w = np.empty_like(zlocs, dtype=object)
    A_t = np.empty_like(zlocs, dtype=object)
    for j in range( len(zlocs) ):
        espectro_z = np.zeros( len(ysol) )*(1 + 0j)
        for i in range( len(ysol) ):
            espectro_z[i] = ysol[i][j]
        A_w[j] = espectro_z * np.exp( D_w * zlocs[j])
        A_t[j] = IFT( A_w[j] )
    return zlocs, A_w, A_t
```

solvebarrierNLSE.py

Solve_barrierNLSE

```
def Solve_barrierNLSE(sim: Sim, fib: Fibra, pulso_0, delta_beta1, TB, betab, z_locs=None,
↪  pbar=True):
#Calculamos el espectro inicial, es lo que vamos a evolucionar.
espectro_0 = FT(pulso_0)
#Calculamos el operador lineal
D_w = 1j* delta_beta1 * (2*np.pi*sim.freq) + 1j * fib.beta2/2 * (2*np.pi*sim.freq)**2 + 1j *
↪  fib.beta3/6 * (2*np.pi*sim.freq)**3 - fib.alpha/2
D_w = np.array(D_w)
if fib.betas:
    D_w = 1j*delta_beta1 * (2*np.pi*sim.freq)
    for i in range(len(fib.betas)):
        D_w = D_w + 1j*fib.betas[i]/np.math.factorial(i+2) * (2*np.pi*sim.freq)**(i+2)
    D_w = np.array(D_w)
#Construimos preliminares: Función de Heaviside
heavi = 1j * betab * np.heaviside(sim.tiempo - TB,1)
#Introducimos todos los parametros en la función, quedando f(z, B) = dB/dz
f_B = partial(dBdz, D = D_w, w = 2*np.pi*sim.freq, gamma=fib.gamma, heavi=heavi)
#Tolerancias para integrar (Tolerancias estandar: rtol=1e-5, atol=1e-8)
rtol = 1e-3
atol = 1e-6
#Usamos solve_ivp: Buscamos solución entre 0 y L
if pbar:
    with tqdm(total=fib.L, unit="m") as pbar:
        def dBdz_with_progress(z, B):
            pbar.update(abs(z - dBdz_with_progress.prev_z))
            dBdz_with_progress.prev_z = z
            return dBdz(z, B, D_w, 2*np.pi*sim.freq, fib.gamma, heavi)
        dBdz_with_progress.prev_z = 0

        if z_locs:
            t_eval = np.linspace(0,fib.L,z_locs)
            sol = solve_ivp(dBdz_with_progress, [0, fib.L], y0 = espectro_0, rtol = rtol, atol =
            ↪  atol, t_eval=t_eval)
        else:
            sol = solve_ivp(dBdz_with_progress, [0, fib.L], y0 = espectro_0, rtol = rtol, atol =
            ↪  atol)
else:
    if z_locs:
        t_eval = np.linspace(0,fib.L,z_locs)
        sol = solve_ivp(lambda z, B: dBdz(z, B, D_w, 2*np.pi*sim.freq, fib.gamma, heavi), [0,
        ↪  fib.L], y0 = espectro_0, rtol = rtol, atol = atol, t_eval=t_eval)
    else:
        sol = solve_ivp(lambda z, B: dBdz(z, B, D_w, 2*np.pi*sim.freq, fib.gamma, heavi), [0,
        ↪  fib.L], y0 = espectro_0, rtol = rtol, atol = atol)
zlocs = sol["t"]  #Puntos de z donde tenemos B(w,z)
ysol  = sol["y"]  #Array, en cada elemento se tiene un subarray [B(w0,z0), B(w0,z1), ...,
↪  B(w0,zf)]
print(sol["message"])
#Armamos array de arrays A(w) y A(t).
ysol  = np.array(ysol)
ysol_transposed = ysol.T
A_w = ysol_transposed
for j in range( len(zlocs) ):
    A_w[j,:] = A_w[j,:] * np.exp(D_w * zlocs[j])
A_t = np.array([IFT(a_w) for a_w in A_w], dtype=complex)
return zlocs, A_w, A_t, ysol
```

Librerias

Este código usa las siguientes funciones y librerias:

NumPy

SciPy

- scipy.fftpack: fft, ifft, fftshift, ifftshift
- scipy.integrate.solve_ivp

functools

- functools.partial

Matplotlib

- matplotlib.pyplot: plot, contourf, specgram