laserfun

laserfun developers

Jun 15, 2023

CONTENTS:

Start by having a look at the *README*.

CHAPTER

ONE

LASERFUN README

Some fun functions for modeling laser pulses in Python.

Documentation is available at: laserfun.readthedocs.io

1.1 Introduction

So far, `laserfun` consists mainly of:

- The Pulse class, which handles the amplitude of the electic field of the pulse in the time and frequency domains.
- The Fiber class, which keeps track of the properties of the fiber, including the dipersion, nonlinearity, length, loss, etc. (By "fiber", we refer to any medium where it is appropriate to model a laser pulse in a single spatial mode which doesnt change over the propagation length. So, optical fibers and optical waveguides would be the most appropriate. Short distances of free-space propagation may also be appropriate, but important effects such as diffraction are ignored.)
- The NLSE function, which models the propagation of a pulse object through a fiber object according to the generalized nonlinear Schrodinger equation (GNLSE) as described in "Supercontinuum Generation in Optical Fibers" Edited by J. M. Dudley and J. R. Taylor (Cambridge 2010).The GNLSE propagates an optical input field (a laser pulse) through a nonlinear material (a fiber) and takes into account dispersion and Chi-3 nonlinearity.

1.2 Intallation

1.2.1 Requirements

laserfun requires Python 3.9+. NumPy and SciPy (version 1.6.0+) are also required, and Matplotlib is required to run the examples. If you don't already have Python, we recommend an "all in one" Python package such as the Anaconda Python Distribution, which is available for free.

1.2.2 With pip

Sorry, were not on PyPi just yet. But soon!

1.2.3 With setuptools

If you might contribute to the laserfun project, we recommend that you fork the repository to your own account, click "open in GitHub desktop", and save your fork somewhere on your computer. Then, navigate to the laserfun folder on the command line (Anaconda prompt in Windows or Terminal on the Mac) and type

```
python setup.py develop
```

This method of installation allows you to modify the source code in-place without re-installing each time.

If you just want to install the code, then you can simply download this repository as a zip file, extract it, navigate to the laserfun folder on the command line, and type

```
python setup.py install
```

1.3 Example of use

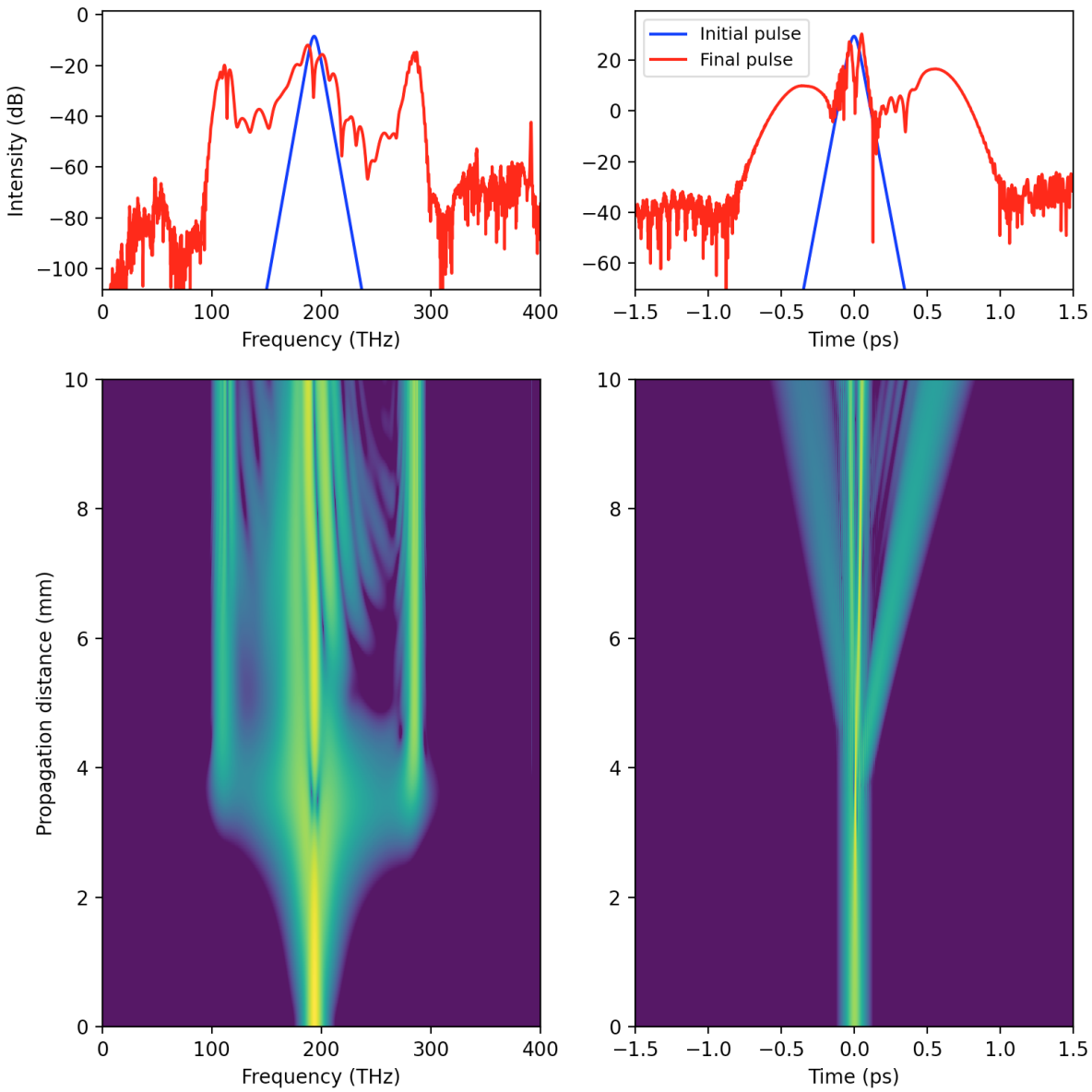
Here is a basic example that generates a pulse object using a 50-fs sech function, creates a fiber object with some dispersion, and propagates the pulse through the fiber using the NLSE.

```
import laserfun as lf

pulse = lf.Pulse(pulse_type='sech', fwhm_ps=0.050, epp=50e-12, center_wavelength_nm=1550)
fiber1 = lf.Fiber(length=0.010, center_wl_nm=1550, dispersion=(-0.12, 0, 5e-6), gamma_W_
↪m=1)
results = lf.NLSE(pulse, fiber1)

results.plot()
```

Here is the output:



Note: Additional examples are located in the *examples* directory.

1.4 Contributing

The laserfun project welcomes suggestions and pull request! The best place to start is to open a new Issue here: https://github.com/laserfun/laserfun/issues.

The following subsections contain a few notes for developers.

1.4.1 Unit tests

To run the tests, go to the PyNLSE folder and run:

```
pytest nlse -v --cov=nlse
```

Coverage can be checked with:

```
coverage html
```

which generates a html file that shows which lines are covered by the tests.

1.4.2 Building documentation

To build the documentation, go to the laserfun/doc folder on the command line and type:

make html

Then, you can open

laserfun/doc/build/html/index.html

in a web browser to view the documentation.

1.5 License

laserfun is distributed under the MIT License.

Enjoy!

CHAPTER

TWO

LASERFUN PACKAGE

2.1 laserfun.nlse module

Functions related to propagation of pulses according to the NLSE.

laserfun.nlse.**NLSE**(*pulse*, *fiber*, *nsaves=200*, *atol=0.0001*, *rtol=0.0001*, *reload_fiber=False*, *raman=False*, *shock=True*, *integrator='lsoda'*, *print_status=True*)

Propagate an laser pulse through a fiber according to the NLSE.

This function propagates an optical input field (often a laser pulse) through a nonlinear material using the generalized nonlinear Schrodinger equation, which takes into account dispersion and nonlinearity. It is a "one dimensional" calculation, in that it doesn't capture things like self focusing and other geometric effects. It's most appropriate for analyzing light propagating through optical fibers.

This code is based on the Matlab code found at www.scgbook.info, which is based on Eqs. (3.13), (3.16) and (3.17) of the book "Supercontinuum Generation in Optical Fibers" Edited by J. M. Dudley and J. R. Taylor (Cambridge 2010). The original Matlab code was written by J.C. Travers, M.H. Frosz and J.M. Dudley (2009). They ask that you cite the book in publications using their code.

Parameters

- **pulse** (*pulse object*) – This is the input pulse.
- **fiber** (*fiber object*) – This defines the media ("fiber") through which the pulse propagates.
- **nsaves** (*int*) – The number of equidistant grid points along the fiber to return the field. Note that the integrator usually takes finer steps than this, the nsaves parameters simply determines what is returned by this function.
- **atol** (*float*) – Absolute tolerance for the integrator. Smaller values produce more accurate results but require longer integration times. 1e-4 works well.
- **rtol** (*float*) – Relative tolerance for the integrator. 1e-4 work well.
- **reload_fiber** (*boolean*) – This determines if the fiber information is reloaded at each step. This should be set to True if the fiber properties (gamma, dispersion) vary as a function of length.
- **raman** (*boolean*) – Determines if the Raman effect will be included. Default is False.
- **shock** (*boolean*) – Determines if the self-steepening (shock) term will be taken into account. This is especially important for situations where the slowly varying envelope approximation starts to break down, which can occur for large bandwidths (short pulses).
- **integrator** (*string*) – Selects the integrator that will be passes to scipy.integrate.ode. options are 'lsoda' (default), 'vode', 'dopri5', 'dopri853'. 'lsoda' is a good option, and seemed

fastest in early tests. I think 'dopri5' and 'dopri853' are simpler Runge-Kutta methods, and they seem to take longer for the same result. 'vode' didn't seem to produce good results with "method='adams'", but things werereasonable with "method='bdf'" For more information, see: docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.ode.html

- **print_status** (*boolean*) – This determines if the propagation status will be printed. Default is True.

Returns

results – This object contains all of the results. Use `z, f, t, AW, AT = results.get_results()` to unpack the z-coordinates, frequency grid, time grid, amplitude at each z-position in the freuqency domain, and amplitude at each z-position in the time domain.

Return type

PulseData object

class laserfun.nlse.**PulseData**(*z*, *AW*, *AT*, *pulse_in*, *pulse_out*, *fiber*)

Bases: `object`

Process data from a pulse propagation.

The following list of parameters is the attributes of the class.

Parameters

- **z** (*array of length n*) – The z-values corresponding the the propagation.
- **f_THz** (*array of length m*) – The values of the frequency grid in THz.
- **t_ps** (*array of length m*) – The values of time grid in ps
- **AT** (*2D array*) – The complex amplitude of the electric field in the time domain at each position in z.
- **AW** (*2D array*) – The complex amplitude of the electric field in the freq. domain at each position in z.
- **pulse_in** (*pulse object*) – The input pulse object.
- **pulse_out** (*pulse object*) – The output pulse object.
- **fiber** (*fiber object*) – The fiber object that the pulse was propagated through.

calc_coherence(*pulse_in*, *fiber*, *num_trials=5*, *n_steps=100*, *random_seed=None*, *noise_type='one_photon_freq'*, ***nlse_kwargs*)

This function runs several nlse simulations (given by num_trials), each time adding random noise to the pulse. By comparing the electric fields of the different pulses, an estimate of the coherence can be made.

Parameters

- **pulse_in** (*pulse object*) –
- **num_trials** (*int*) – this determines the number of trials to be run.
- **random_seed** (*int*) – this is the seed for the random noise generation. Default is None, which does not set a seed for the random number generator, which means that the numbers will be completely randomized. Setting the seed to a number (i.e., random_seed=0) will still generate random numbers for each trial, but the results from calculate_coherence will be completely repeatable.
- **noise_type** (*str*) – this specifies the method for including random noise onto the pulse. See `pynlo.light.PulseBase.Pulse.add_noise()` for the different methods.

Returns

- **g12W** (*2D numpy array*) – This 2D array gives the g12 parameter as a function of propagation distance and the frequency. g12 gives a measure of the coherence of the pulse by comparing several different trials.
- **results** (*list of results for each trial*) – This is a list, where each item of the list contains (z_positions, AW, AT, pulse_out), the results obtained from `pynlo.interactions.FourWaveMixing.SSFM.propagate()`.

`get_results`(*data_type='amplitude'*, *rep_rate=1*)

Get the frequency domain (AW) and time domain (AT) results of the NLSE propagation. Also provides the length (z), frequnecy (f), and time (t) arrays.

`'amplitude'` - Native units for the NLSE, AT and AW are sqrt(W). Does NOT consider the rep-rate.

`'intensity'` - Absolute value of amplitude squared. These units make some sense for AT, since they are J/sec, so integrating over the pulse works as expected. Units for AW are J*Hz, so be careful when integrating. Does NOT consider rep-rate.

`'mW/bin'` - AW and AT are in mW per bin, so naive summing provides average power (in mW). Rep-rate taken into account.

`'mW/THz'` - returns AW in units of mW/THz and AT in mW/ps. Rep-rate taken into account.

`'dBm/THz'` - returns AW in units of mW/THz and AT in dBm/ps. Rep-rate taken into account.

`'mW/nm'` - returns AW in units of mW/nm and AT in mW/ps. Rep-rate taken into account.

`'dBm/nm'` - returns AW in units of dBm/nm and AT in dBm/ps. Rep-rate taken into account.

In the above, dBm is 10*log10(mW).

Note that, for the "per nm" situations, AW is still returned on a grid of even *frequencies*, so the uneven wavelength spacing should be taken into account when integrating. Use `get_results_wavelength` to re-interpolate to an evenly spaced wavelength grid.

In order to get per-pulse numbers for all methods, simple set the rep- rate to 1.

Parameters

- **data_type** (*'string'*) – Determines the units for the returned AW and AT arrays.
- **rep_rate** (*float*) – The repetition rate of the pulses for calculation of average power units. Does not affect the "amplitude" or "intensity" calculations, but scales all other calculations.

Returns

- **z** (*1D numpy array of length nsaves*) – Array of the z-coordinate along fiber, units of meters.
- **f_THz** (*1D numpy array of length n.*) – The frequency grid (not angular freq) in THz.
- **t_ps** (*1D numpy array of length n*) – The temporal grid in ps.
- **AW** (*2D numpy array, with dimensions nsaves x n*) – The complex amplitide of the frequency domain field at every step.
- **AT** (*2D numpy array, with dimensions nsaves x n*) – The complex amplitude of the time domain field at every step.

`get_results_wavelength`(*wmin=None*, *wmax=None*, *wn=None*, *data_type='intensity'*, *rep_rate=1*)

Get results on a wavelength grid.

Re-interpolates the AW array from evenly-spaced frequencies to evenly-spaced wavelengths.

Parameters

- **wmin** (*float or None*) – the minimum wavelength for the re-interpolation grid. If None, it defaults to 0.25x the center wavelength of the pulse. If a float, this is the minimum wavelength of the pulse in nm
- **wmax** (*float or None*) – the minimum wavelength for the re-interpolation grid. If None, it defaults to 4x the center wavelength of the pulse. If a float, this is the maximum wavelength of the pulse in nm
- **wn** (*int or None*) – number of wavelengths for the re-interpolation grid. If None, it defaults to the number of points in AW multiplied by 2 If an int, then this is just the number of points.
- **data_type** (*'string'*) – Determines the units for the AW and AT arrays. See the documentation for the `get_results` function for more information. Note that `data_type='amplitude'` is supported but not recommended because interpolation on the rapidly varying grid of complex values can lead to inconsistent results.
- **rep_rate** (*float*) – The repetition rate of the pulses for calculation of average power units. Does not affect the "amplitude" or "intensity" calculations, but scales all other calculations.

Returns

- **z** (*1D numpy array of length nsaves*) – Array of the z-coordinate along fiber.
- **wls** (*1D numpy array of length n.*) – The wavelength grid.
- **t** (*1D numpy array of length n*) – The temporal grid.
- **AW_WLS** (*2D numpy array, with dimensions nsaves x n*) – The complex amplitide of the frequency domain field at every step.
- **AT** (*2D numpy array, with dimensions nsaves x n*) – The complex amplitude of the time domain field at every step.

plot(*flim=30*, *tlim=50*, *margin=0.2*, *wavelength=False*, *show=True*, *units='intensity'*, *rep_rate=100000000.0*)

Plot the results in both the time and frequency domain.

Parameters

- **flim** (*float or array of length 2*) – This sets the xlimits of the frequency domain plot. If this is a single number, it defines the dB level at which the plot will be set (with a margin). If an array of two values, this manually sets the xlims.
- **tlim** (*float or array of length 2*) – Same as flim, but for the time domain.
- **margin** (*float*) – Fraction to pad the xlimits. Default is 0.2.
- **wavelength** (*boolean*) – Determines if the "frequency" domain will be displayed in Frequency (THz) or wavelength (nm).
- **show** (*boolean*) – determines if plt.show() will be called to show the plot
- **units** (*string*) – Units for the frequency-domain plots. For example, dBm/THz mW/THz. See the documentation for the `data_type` keyword argument for the `get_results` method for more information.
- **rep_rate** (*float*) – The repetition rate of the pule train in Hz. This is used to calculate the average powers when using units other than "intensity".

Returns

- **fig** (*matplotlib.figure object*) – The figure object so that modifications can be made.

- **axs** (*an 2x2 array of axes objects*) – The axes objects, so that modifications can be made. For example: axs[0, 1].set_xlim(0, 1000)

laserfun.nlse.**dB**(*num*)

2.2 laserfun.pulse module

Tools for working with laser pulses.

laserfun.pulse.**FFT_t**(*A*, *ax=0*)

Do a FFT with fft-shifting.

laserfun.pulse.**IFFT_t**(*A*, *ax=0*)

Do an iFFT with fft-shifting.

class laserfun.pulse.**Pulse**(*pulse_type='sech'*, *center_wavelength_nm=1550*, *fwhm_ps=0.2*, *time_window_ps=10.0*, *power=1*, *epp=None*, *npts=4096*, *power_is_avg=False*, *frep_MHz=100*, *GDD=0*, *TOD=0*, *FOD=0*)

Bases: `object`

Generate a new pulse object based on a pre-defined pulse shapes.

Customized pulses can be generated by calling this function and then manually setting the time- or frequency domain pulse-profile using `pulse.at` or `pulse.aw`.

Note that the pulse object is intrinsically a single laser pulse, and the time and frequency domain electric fields refer to the single pulse, not the average power of the laser at some repetition rate. The ` *power_is_avg* ` and `frep_MHz` values are used only to set the electric field during the generation of the pulse and are not stored in the pulse object.

Parameters

- **pulse_type** (*string*) – The shape of the pulse in the time domain. Options are:
  - sech, which produces a hyperbolic secant (sech) shaped pulse `A(t) = sqrt(power) * sech(t/T0)`, where `T0=fwhm/1.76`
  - gaussian, which produces a Gaussian shaped pulse `A(t) = sqrt(power) * exp(-(t/T0)^2/2)`, where `T0=fwhm/1.76`
  - sinc, which uses a sin(x)/x (sinc) function `A(t) = sqrt(power) * sin(t/T0)/(t/T0)`, were `T0=fwhm/3.79`
- **center_wavelength_nm** (*float*) – The center wavelength of the pulse in nm.
- **fwhm_ps** (*float*) – The full-width-at-half-maximum of the pulse in picoseconds.
- **time_window_ps** (*float*) – The time window in picoseconds. This is the full-width of the time window, so the times will go from `-time_window_ps * 0.5` to `+ time_window_ps * 0.5`.
- **power** (*float*) – this is either the peak power of the pulse or the average power of the pulse-train, depending on `power_is_avg`. In both cases the units are in watts.
- **epp** (*float or None*) – the energy-per-pulse in Joules. If this is not None (the default), then this overrides the power argument to set the pulse energy.
- **npts** (*int*) – the number of points in the time and frequency grids. Using powers of 2 might be beneficial for the efficiency of the FFTs in the NLSE algorithm. `2**12` is a good starting point.

- **power_is_avg** (*boolean*) – determines if the power is peak power or average power. Note that this value is only used for calculating the initial pulse amplitude and is not saved as an intrinsic characteristic of the pulse object.
- **frep_MHz** (*float*) – the repetition rate in MHz. Used for setting the peak power of the pulse to match the average power of the pulse train. Similar to power_is_avg, the rep rate is not saved as part of the pulse class.
- **GDD** (*float*) – the group-delay-dispersion (in ps^2) to apply to the pulse.
- **TOD** (*float*) – the third-order dispersion (in ps^3) to apply to the pulse.
- **FOD** (*float*) – the fourth-order dispersion (in ps^3) to apply to the pulse.

add_noise(*noise_type='sqrt_N_freq'*)

Add random intensity and phase noise to a pulse.

Parameters
noise_type (*string*) – The method used to add noise. The options are:

- sqrt_N_freq : adds noise to each bin in the frequency domain. The average noise added is proportional to sqrt(N), and where N is the number of photons in that frequency bin.
- one_photon_freq` : which adds one photon of noise to each frequency bin.

property at

Amplitude of the time-domain electric field (complex).

Units are sqrt(W), which can be considered sqrt(J)/sqrt(s), so units of energy per time bin when the absolute value is squared.

property aw

Set/get the complex amplitude of the pulse in the frequency domain.

Corresponds to the frequencies in the pulse.aw array. Note that this is the complex amplitude and that the intensity is the square of the absolute value.

The units are sqrt(W) or sqrt(J)*sqrt(Hz), so abs(aw)^2 * deltaF will provide J/bin. (90% sure this comment about units is correct.)

calc_width(*level=0.5*)

Calculate the pulse width.

For example, the full-width-at-half-maxmimum (FWHM) or the 1/e width. If the pulse has multiple crossings (for example, if it reaches the 0.5 level multiple times) the widest extent will be returned.

Parameters
level (*float*) – the fraction of the peak to calculate the width. Must be between 0 and 1. Default is 0.5, which provides the FWHM. 1/e = 0.367879 1/e^2 = 0.135335

Returns
width – the width of the pulse in picoseconds.

Return type
float

property center_wavelength_nm

Set/get the center wavelength of the grid in units of nanometers.

property centerfrequency_THz

Set/get The center frequency (float) of the pulse in THz.

chirp_pulse_W(*GDD*, *TOD=0*, *FOD=0.0*, *w0_THz=None*)

Alter the phase of the pulse.

Apply the dispersion coefficients $\beta_2, \beta_3, \beta_4$ expanded around frequency ω_0.

Parameters

- **GDD** (*float*) – Group delay dispersion (β_2) [ps^2]
- **TOD** (*float, optional*) – Group delay dispersion (β_3) [ps^3], defaults to 0.
- **FOD** (*float, optional*) – Group delay dispersion (β_4) [ps^4], defaults to 0.
- **w0_THz** (*float, optional*) – Center freq. of dispersion expansion, defaults to grid center freq.

Notes

The convention used for dispersion is

$$E_{new}(\omega) = \exp\left(i\left(\frac{1}{2}GDD\,\omega^2 + \frac{1}{6}TOD\omega^3 + \frac{1}{24}FOD\,\omega^4\right)\right)E(\omega)$$

clone_pulse(*p*)

Copy all parameters of pulse_instance into this one.

create_cloned_pulse()

Create and return new pulse instance identical to this instance.

property df_THz

Frequency grid spacing in THz.

property dt_ps

Return time grid spacing in ps.

property epp

Energy per pulse in Joules.

property f_THz

Get the absolute frequency grid in THz.

property npts

Set/get Number of points (int) in the time (and frequency) grid.

psd(*units='W'*, *rep_rate=1*)

Return the power spectral density (PSD) in various units. Set the rep_rate to 1 to get "per pulse" units. Otherwise, the rep-rate will be used to scale the per-pulse numbers to average power units. By default, the rep-rate of the pulse object is used.

Unit options are:

`'mW'`, mW for each data point. (Not actually PSD units.) `'mW/THz'`, mW per THz. `'dBm/THz'`, 10*log10(mW) per THz. `'mW/nm'`, mW per nanometer. `'dBm/nm'`, 10*log10(mW) per nanometer.

Note that for the "per nanometer" units, the data is still delivered on a grid that is evenly spaced in *frequency*, not wavelength. So, if integrating the PSD, it is necessary to take into account the changing size of the wavelength bins to recover the correct value for the average power. The psd_wavelength function provides both the evenly spaced wavelength grid and the y-axis unit conversion.

Parameters

- **units** (*str*) – Determines the units of the intensity of power-spectral-density. See above for options.
- **rep_rate** (*float*) – Determines the repetition rate (in Hz) used to calculate the PSD. Set to 1 to get per-pulse PSD. If None, then the rep-rate for the pulse object will be used. Note that the rep-rate provided to this function doesn't change the rep-rate of the pulse object, it is merely used for the PSD calculation.

Returns
psd – numpy array of power spectral densities corresponding to pulse.aw.

Return type
array

psd_wavelength(*wl_min=500*, *wl_max=2500*, *wl_step=0.2*)

– Not yet implemented – evenly spaced wavelength and intensity in various units

property t_ps

Get the temporal grid in ps.

property time_window_ps

Set/get the time window of the time grid in picoseconds (float).

transform_limit()

"Return a transform-limited pulse (flat spectral phase).

property v_THz

Get the *relative angular* frequency grid in THz.

property w0_THz

Get the center *angular* frequency (THz).

property w_THz

Get the absolute *angular* frequency grid (THz).

property wavelength_nm

Wavelength grid in nanometers.

2.3 laserfun.fiber module

Contains classes and functions for working with fibers.

class laserfun.fiber.**Fiber**(*length=0.1*, *center_wl_nm=1550*, *dispersion_format='GVD'*, *dispersion=[0]*, *gamma_W_m=0*, *loss_dB_per_m=0*)

Bases: object

A class that contains the information about a fiber.

get_B(*pulse*, *z=0*)

Get the propagation constant (Beta) at the frequency grid of pulse.

Here, B refers to the propagation constant, in units of 1/meters, and not to the Beta2 parameter. Also, these are relative B values, in that they are set to zero at the pulse central frequency. Referring to these values as "integrated dispersion" might be appropriate.

Three different methods are used,

If fiberspecs["dispersion_format"] == "D", then the DTabulationToBetas function is used to fit the data-points in terms of the Beta2, Beta3, etc. coefficients expanded around the pulse central frequency.

If fiberspecs["dispersion_format"] == "GVD", then the betas are calculated as a Taylor expansion using the Beta2, Beta3, etc. coefficients around the *fiber* central frequency.

If fiberspecs["dispersion_format"] == "n", then the betas are calculated directly from the **effective refractive index (n_eff)** as `beta = n_eff * 2 * pi / lambda`, where lambda is the wavelength of the light. In this case, self._x should be the wavelength (in nm) and self._y should be n_eff (unitless).

Parameters

- **pulse** (*pulse object*) – the pulse object must be supplied to define the frequency grid.
- **z** (*float*) – the postion along the length of the fiber. The units of this must match the units expected by the functions provided to set_dispersion_function() and set_gamma_function(). Just use meters!

Returns
B – the propagation constant (beta) at the frequency gridpoints of the supplied pulse (units of 1/meters).

Return type
1D array of floats

get_alpha(*z=0*)

Query alpha (loss per meter) at a specific z-position.

Parameters
z (*float*) – the position along the fiber (in meters)

Returns
alpha – the loss (in units of 1/Watts)

Return type
float

get_gamma(*z=0*)

Query gamma (effective nonlinearity) at a specific z-position.

Parameters
z (*float*) – the position along the fiber (in meters)

Returns
gamma – the effective nonlinearity (in units of 1/(Watts * meters))

Return type
float

set_alpha_function(*gamma_function*)

Set alpha (loss) function that varies with z.

Gamma should be in units of 1/(W m), NOT 1/(W km)

Parameters
alpha_function (*function*) – a function returning gamma as a function of z. z should be in units of meters.

set_dispersion_function(*dispersion_function*, *dispersion_format='GVD'*)

Set dispersion to function that varies as a function of z.

The function can either provide beta2, beta3, beta4, etc. coefficients, or provide two arrays, wavelength (nm) and D (ps/nm/km).

Parameters

- **dispersion_function** (*function*) – A function returning D or Beta coefficients as a function of z. z should be in meters.
- **dispersion_format** (*'GVD' or 'D' or 'n'*) – determines if the dispersion will be in terms of Beta coefficients (GVD, in units of ps^2/m, not ps^2/km), D (ps/nm/km), or n (effective refractive index)

set_gamma_function(*gamma_function*)

Set gamma to a function that varies as a function of z.

The function should return gamma (the effective nonlinearity) in units of 1/(Watts * meters)) that can vary as a function of *z*, the length along the fiber.

Parameters

gamma_function (*function*) – a function returning gamma as a function of z. z should be in units of meters.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

W