PyNLO Documentation

Release 0.1

Gabriel Ycas

Oct 25, 2018

Contents

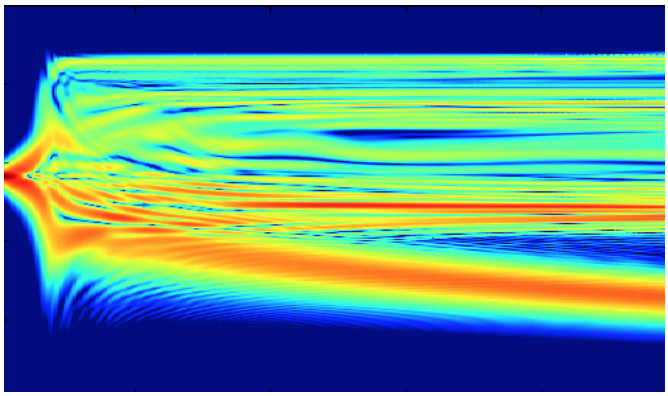
Contents:

CHAPTER 1

pyNLO: Nonlinear optics modeling for Python

This README is best viewed at http://pynlo.readthedocs.io/en/latest/readme_link.html

Complete documentation is available at http://pynlo.readthedocs.io/



1.1 Introduction

PyNLO provides an easy-to-use, object-oriented set of tools for modeling the nonlinear interaction of light with materials. It provides many functionalities for representing pulses of light, beams of light, and nonlinear materials, such as crystals and fibers. Also, it features methods for simulating both three-wave-mixing processes (such as DFG), as well as four-wave-mixing processes such as supercontinuum generation.

Features:

- A solver for the propagation of light through a Chi-3 material, useful for simulation pulse compression and supercontinuum generation in an optical fiber. This solver is highly efficient, thanks to an adaptive-step-size implementation of the "Fourth-order Runge-Kutta in the Interaction Picture " (RK4IP) method of Hult (2007).
- A solver for simulating Chi-2 processes such as difference frequency generation.

- A flexible object-oriented system for treating laser pulses, beams, fibers, and crystals.
- . . . and much more!

1.2 Installation

PyNLO requires Python 2, and is tested on Python 2.7 (Python 3 compatibility is a work-in-progress). 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.

1.2.1 With pip

The latest "official release" can be installed from PyPi with

```
pip install pynlo
```

The up-to-the-minute latest version can be installed from GitHub with

```
pip install git+https://github.com/pyNLO/PyNLO.git
```

1.2.2 With setuptools

Alternatively, you can download the latest version from the PyNLO Github site (look for the "download zip" button), *cd* to the PyNLO directory, and use

```
python setup.py install
```

Or, if you wish to edit the PyNLO source code without re-installing each time

```
python setup.py develop
```

1.3 Documentation

The complete documentation for PyNLO is availabe at https://pynlo.readthedocs.org.

1.4 Example of use

The following example demonstrates how to use PyNLO to simulate the propagation of a 50 fs pulse through a nonlinear fiber using the split-step Fourier model (SSFM). Note that the actual propagation of the pulse takes up just a few lines of code. Most of the other code is simply plotting the results.

This example is contained in examples/simple_SSFM.py

```
import numpy as np
import matplotlib.pyplot as plt
import pynlo

FWHM    = 0.050  # pulse duration (ps)
```

(continues on next page)

(continued from previous page)

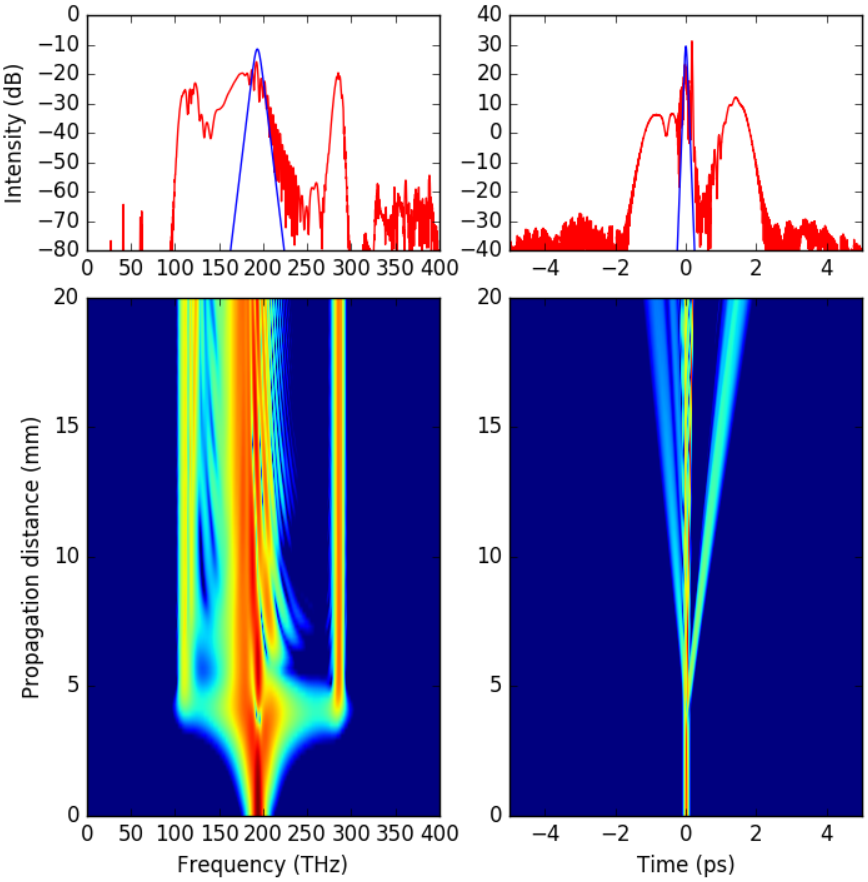
```
pulseWL = 1550    # pulse central wavelength (nm)
EPP     = 50e-12  # Energy per pulse (J)
GDD     = 0.0     # Group delay dispersion (ps^2)
TOD     = 0.0     # Third order dispersion (ps^3)

Window  = 10.0    # simulation window (ps)
Steps   = 100      # simulation steps
Points  = 2**13   # simulation points

beta2   = -120      # (ps^2/km)
beta3   = 0.00      # (ps^3/km)
beta4   = 0.005     # (ps^4/km)

Length  = 20     # length in mm

Alpha   = 0.0      # attentuation coefficient (dB/cm)
Gamma   = 1000     # Gamma (1/(W km)

fibWL   = pulseWL # Center WL of fiber (nm)

Raman   = True     # Enable Raman effect?
Steep   = True     # Enable self steepening?

alpha = np.log((10**(Alpha * 0.1))) * 100  # convert from dB/cm to 1/m


# set up plots for the results:
fig = plt.figure(figsize=(8,8))
ax0 = plt.subplot2grid((3,2), (0, 0), rowspan=1)
ax1 = plt.subplot2grid((3,2), (0, 1), rowspan=1)
ax2 = plt.subplot2grid((3,2), (1, 0), rowspan=2, sharex=ax0)
ax3 = plt.subplot2grid((3,2), (1, 1), rowspan=2, sharex=ax1)


######## This is where the PyNLO magic happens! ############################

# create the pulse!
pulse = pynlo.light.DerivedPulses.SechPulse(power = 1, # Power will be scaled by set_
↪epp
                                            T0_ps                   = FWHM/1.76,
                                            center_wavelength_nm    = pulseWL,
                                            time_window_ps          = Window,
                                            GDD=GDD, TOD=TOD,
                                            NPTS            = Points,
                                            frep_MHz        = 100,
                                            power_is_avg    = False)
# set the pulse energy!
pulse.set_epp(EPP)

# create the fiber!
fiber1 = pynlo.media.fibers.fiber.FiberInstance()
fiber1.generate_fiber(Length * 1e-3, center_wl_nm=fibWL, betas=(beta2, beta3, beta4),
                              gamma_W_m=Gamma * 1e-3, gvd_units='ps^n/km', gain=-
↪alpha)

# Propagation
evol = pynlo.interactions.FourWaveMixing.SSFM.SSFM(local_error=0.005, USE_SIMPLE_
↪RAMAN=True,
```

(continues on next page)

(continued from previous page)

```
                disable_Raman             = np.logical_not(Raman),
                disable_self_steepening   = np.logical_not(Steep))

y, AW, AT, pulse_out = evol.propagate(pulse_in=pulse, fiber=fiber1, n_steps=Steps)

########## That's it! Physics complete. Just plotting commands from here! ############
↪####


F = pulse.F_THz     # Frequency grid of pulse (THz)

def dB(num):
    return 10 * np.log10(np.abs(num)**2)

zW = dB( np.transpose(AW)[:, (F > 0)] )
zT = dB( np.transpose(AT) )

y_mm = y * 1e3 # convert distance to mm

ax0.plot(pulse_out.F_THz,    dB(pulse_out.AW),  color = 'r')
ax1.plot(pulse_out.T_ps,     dB(pulse_out.AT),  color = 'r')

ax0.plot(pulse.F_THz,    dB(pulse.AW),  color = 'b')
ax1.plot(pulse.T_ps,     dB(pulse.AT),  color = 'b')

extent = (np.min(F[F > 0]), np.max(F[F > 0]), 0, Length)
ax2.imshow(zW, extent=extent,
           vmin=np.max(zW) - 40.0, vmax=np.max(zW),
           aspect='auto', origin='lower')

extent = (np.min(pulse.T_ps), np.max(pulse.T_ps), np.min(y_mm), Length)
ax3.imshow(zT, extent=extent,
           vmin=np.max(zT) - 40.0, vmax=np.max(zT),
           aspect='auto', origin='lower')


ax0.set_ylabel('Intensity (dB)')
ax0.set_ylim( - 80,  0)
ax1.set_ylim( - 40, 40)

ax2.set_ylabel('Propagation distance (mm)')
ax2.set_xlabel('Frequency (THz)')
ax2.set_xlim(0,400)

ax3.set_xlabel('Time (ps)')

plt.show()
```

Here are the results:



1.5 Contributing

We welcome suggestions for improvement, questions, comments, etc. The best way to to open a new issue here: https://github.com/pyNLO/PyNLO/issues/.

1.6 License

PyNLO is licensed under the GPLv3 license. This means that you are free to use PyNLO for any **open-source** project. Of course, PyNLO is provided "as is" with absolutely no warrenty.

1.7 References

[1] Johan Hult, "A Fourth-Order Runge–Kutta in the Interaction Picture Method for Simulating Supercontinuum Generation in Optical Fibers," J. Lightwave Technol. 25, 3770-3775 (2007) https://www.osapublishing.org/jlt/abstract.cfm?uri=jlt-25-12-3770

CHAPTER 2

General information on PyNLO

2.1 Package Organization

In pyNLO, object-oriented programming is used to mimic the physics of nonlinear interactions. Whenever possible, each physical entity with intrinic properties – for example an optical pulse or nonlinear fiber – is mapped to a single Python class. These classes keep track of the objects' properties, calculate interactions between them and other objects, and provide simple calculator-type helper functions.

2.2 References

CHAPTER 3

PyNLO package

3.1 pynlo.light

The **light** module contains modules to model light pulses.

3.1.1 pynlo.light.PulseBase

class pynlo.light.PulseBase.**Pulse**(*frep_MHz=None*, *n=None*)

Class which carried all information about the light field. This class is a base upon which various cases are built (eg analytic pulses, CW fields, or pulses generated from experimental data.)

AT

Property – time-domain electric field grid

Returns **AT** – Complex electric field in time domain.

Return type ndarray, shape NPTS

AW

Property – frequency-domain electric field grid

Returns **AW** – Complex electric field in frequency domain.

Return type ndarray, shape NPTS

F_THz

Property – frequency grid

Returns **F_THz** – Frequency grid corresponding to AW [THz]

Return type ndarray, shape NPTS

F_mks

Property – frequency grid

Returns **F_mks** – Frequency grid corresponding to AW [Hz]

Return type ndarray, shape NPTS

T_mks
Property – time grid

Returns **T_mks** – Time grid corresponding to AT [s]

Return type ndarray, shape NPTS

T_ps
Property – time grid

Returns **T_ps** – Time grid corresponding to AT [ps]

Return type ndarray, shape NPTS

V_THz
Property – relative angular frequency grid

Returns **V_THz** – Relative angular frequency grid corresponding to AW [THz]

Return type ndarray, shape NPTS

V_mks
Property – relative angular frequency grid

Returns **V_mks** – Relative angular frequency grid corresponding to AW [Hz]

Return type ndarray, shape NPTS

W_THz
Property – angular frequency grid

Returns **W_THz** – Angular frequency grid corresponding to AW [THz]

Return type ndarray, shape NPTS

W_mks
Property – angular frequency grid

Returns **W_mks** – Angular frequency grid corresponding to AW [Hz]

Return type ndarray, shape NPTS

add_noise(*noise_type='sqrt_N_freq'*)

Adds random intensity and phase noise to a pulse.

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

`sqrt_N_freq` : which adds noise to each bin in the frequency domain, where the sigma is proportional to sqrt(N), and where N is the number of photons in each frequency bin.

`one_photon_freq` : which adds one photon of noise to each frequency bin, regardless of the previous value of the electric field in that bin.

Returns

Return type nothing

add_time_offset(*offset_ps*)
Shift field in time domain by offset_ps picoseconds. A positive offset moves the pulse forward in time.

calc_epp()
Calculate and return energy per pulse via numerical integration of $A^2 dt$

Returns **x** – Pulse energy [J]

Return type float

calculate_intensity_autocorrelation()
Calculates and returns the intensity autocorrelation, $\int P(t)P(t+\tau)dt$

Returns **x** – Intensity autocorrelation. The grid is the same as the pulse class' time grid.

Return type ndarray, shape N_pts

center_frequency_THz
Property – center frequency

Returns **center_frequency_THz** – Frequency of center point in AW grid [THz]

Return type float

center_frequency_mks
Property – center frequency

Returns **center_frequency_mks** – Frequency of center point in AW grid [Hz]

Return type float

center_wavelength_mks
Property – center wavelength

Returns **center_wavelength_mks** – Wavelength of center point in AW grid [m]

Return type float

center_wavelength_nm
Property – center wavelength

Returns **center_wavelength_nm** – Wavelength of center point in AW grid [nm]

Return type float

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 frequency of dispersion expansion, defaults to grid center frequency.

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.

create_subset_pulse(*center_wl_nm*, *NPTS*)
Create new pulse with smaller frequency span, centered at closest grid point to center_wl_nm, with NPTS grid points and frequency-grid values from this pulse.

dF_THz
Property – frequency grid spacing

Returns **dF_ps** – Frequency grid spacing [ps]

Return type float

dF_mks
Property – frequency grid spacing

Returns **dF_mks** – Frequency grid spacing [s]

Return type float

dT_mks
Property – time grid spacing

Returns **dT_mks** – Time grid spacing [s]

Return type float

dT_ps
Property – time grid spacing

Returns **dT_ps** – Time grid spacing [ps]

Return type float

expand_time_window(*factor_log2*, *new_pts_loc='before'*)
Expand the time window by zero padding. :param factor_log2: Factor by which to expand the time window (1 = 2x, 2 = 4x, etc.) :type factor_log2: integer :param new_pts_loc: Where to put the new points. Valid options are "before", "even",

"after

frep_MHz
Property – Repetition rate. Used for calculating average beam power.

Returns **frep_MHz** – Pulse repetition frequency [MHz]

Return type float

frep_mks
Property – Repetition rate. Used for calculating average beam power.

Returns **frep_mks** – Pulse repetition frequency [Hz]

Return type float

interpolate_to_new_center_wl(*new_wavelength_nm*)
Change grids by interpolating the electric field onto a new frequency grid, defined by the new center wavelength and the current pulse parameters. This is useful when grid overlaps must be avoided, for example in difference or sum frequency generation.

Parameters **new_wavelength_nm** (*float*) – New center wavelength [nm]

Returns

Return type Pulse instance

load_consts()
Load constants, needed after unpickling in some cases

rotate_spectrum_to_new_center_wl(*new_center_wl_nm*)
Change center wavelength of pulse by rotating the electric field in the frequency domain. Designed for creating multiple pulses with same gridding but of different colors. Rotations is by integer and to the closest omega.

set_AT(*AT_new*)
Set the value of the time-domain electric field.

Parameters **AW_new** (*array_like*) – New electric field values.

set_AW(*AW_new*)
Set the value of the frequency-domain electric field.

Parameters **AW_new** (*array_like*) – New electric field values.

set_NPTS(*NPTS*)
Set the grid size.

The actual grid arrays are *not* altered automatically to reflect a change.

Parameters **NPTS** (*int*) – Number of points in grid

set_center_wavelength_m(*wl*)
Set the center wavelength of the grid in units of meters.

Parameters **wl** (*float*) – New center wavelength [m]

set_center_wavelength_nm(*wl*)
Set the center wavelength of the grid in units of nanometers.

Parameters **wl** (*float*) – New center wavelength [nm]

set_epp(*desired_epp_J*)
Set the energy per pulse (in Joules)

Parameters **desired_epp_J** (*float*) – the value to set the pulse energy [J]

Returns

Return type nothing

set_frep_MHz(*fr_MHz*)
Set the pulse repetition frequency.

This parameter used internally to convert between pulse energy and average power.

Parameters **fr_MHz** (*float*) – New repetition frequency [MHz]

set_frequency_window_THz(*DF*)
Set the total frequency window of the grid.

This sets the grid dF, and implicitly changes the temporal span (~1/dF).

Parameters **DF** (*float*) – New grid time span [THz]

set_frequency_window_mks(*DF*)
Set the total frequency window of the grid.

This sets the grid dF, and implicitly changes the temporal span (~1/dF).

Parameters **DF** (*float*) – New grid time span [Hz]

set_time_window_ps(*T*)
Set the total time window of the grid.

This sets the grid dT, and implicitly changes the frequency span (~1/dT).

Parameters **T** (*float*) – New grid time span [ps]

set_time_window_s(*T*)
Set the total time window of the grid.

This sets the grid dT, and implicitly changes the frequency span (~1/dT).

Parameters **T** (*float*) – New grid time span [s]

spectrogram(*gate_type='xfrog'*, *gate_function_width_ps=0.02*, *time_steps=500*)
This calculates a spectrogram, essentially the spectrally-resolved cross-correlation of the pulse.

Generally, the gate_type should set to 'xfrog', which performs a cross-correlation similar to the XFROG experiment, where the pulse is probed by a short, reference pulse. The temporal width of this pulse is set by the "gate_function_width_ps" parameter.

See Dudley Fig. 10, on p1153 for a description of the spectrogram in the context of supercontinuum generaiton. (http://dx.doi.org/10.1103/RevModPhys.78.1135)

Alternatively, the gate_type can be set to 'frog', which simulates a SHG-FROG measurement, where the pulse is probed with a copy of itself, in an autocorrelation fashion. Interpreting this FROG spectrogram is less intuitive, so this is mainly useful for comparison with experimentally recorded FROG spectra (which are often easier to acquire than XFROG measurements.)

A nice discussion of various FROG "species" is available here: http://frog.gatech.edu/tutorial.html

Parameters

- **gate_type** (*string*) – Determines the type of gate function. Can be either 'xfrog' or 'frog'. Should likely be set to 'xfrog' unless comparing with experiments. See discussion above. Default is 'xfrog'.
- **gate_function_width** (*float*) – the width of the gate function in seconds. Only applies when gate_type='xfrog'. A shorter duration provides better temporal resolution, but worse spectral resolution, so this is a trade-off. Typically, 0.01 to 0.1 ps works well.
- **time_steps** (*int*) – the number of delay time steps to use. More steps makes a higher resolution spectrogram, but takes longer to process and plot. Default is 500

Returns

- **DELAYS** (*2D numpy meshgrid*) – the columns have increasing delay (in ps)
- **FREQS** (*2D numpy meshgrid*) – the rows have increasing frequency (in THz)
- **spectrogram** (*2D numpy array*) – Following the convention of Dudley, the frequency runs along the y-axis (axis 0) and the time runs alon the x-axis (axis 1)

Example

The spectrogram can be visualized using something like this:

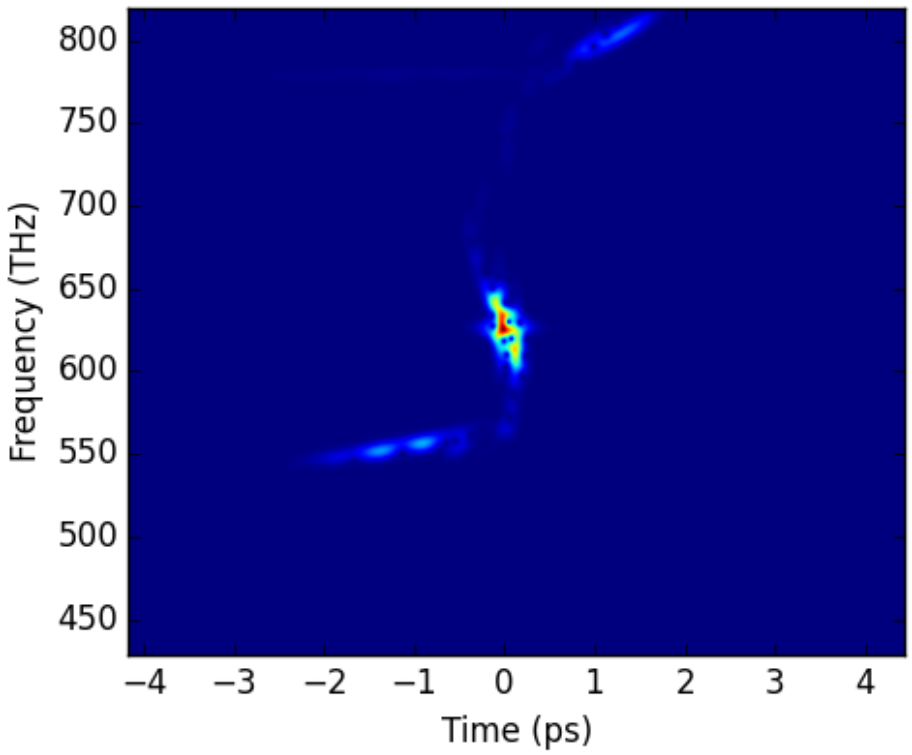
```
import matplotlib.pyplot as plt
plt.figure()
DELAYS, FREQS, extent, spectrogram = pulse.spectrogram()
plt.imshow(spectrogram, aspect='auto', extent=extent)
plt.xlabel('Time (ps)')
```

(continues on next page)

(continued from previous page)

```
plt.ylabel('Frequency (THz)')
plt.tight_layout

plt.show()
```

output:



time_window_mks
Property – time grid span

Returns **time_window_mks** – Time grid span [ps]

Return type float

time_window_ps
Property – time grid span

Returns **time_window_ps** – Time grid span [ps]

Return type float

wl_mks
Property – Wavelength grid

Returns **wl_mks** – Wavelength grid corresponding to AW [m]

Return type ndarray, shape NPTS

wl_nm
Property – Wavelength grid

Returns **wl_nm** – Wavelength grid corresponding to AW [nm]

Return type ndarray, shape NPTS

write_frog(*fileloc='broadened_er_pulse.dat'*, *flip_phase=True*)
Save pulse in FROG data format. Grid is centered at wavelength center_wavelength (nm), but pulse properties are loaded from data file. If flip_phase is true, all phase is multiplied by -1 [useful for correcting direction of time ambiguity]. time_window (ps) sets temporal grid size.

power sets the pulse energy: if power_is_epp is True then the number is pulse energy [J] if power_is_epp is False then the power is average power [W], and is multiplied by frep to calculate pulse energy

3.1.2 pynlo.light.DerivedPulses

class pynlo.light.DerivedPulses.**SechPulse**(*power*, *T0_ps*, *center_wavelength_nm*, *time_window_ps=10.0*, *frep_MHz=100.0*, *NPTS=1024*, *GDD=0*, *TOD=0*, *chirp2=0*, *chirp3=0*, *power_is_avg=False*)

__init__(*power*, *T0_ps*, *center_wavelength_nm*, *time_window_ps=10.0*, *frep_MHz=100.0*, *NPTS=1024*, *GDD=0*, *TOD=0*, *chirp2=0*, *chirp3=0*, *power_is_avg=False*)

Generate a squared-hyperbolic secant "sech" pulse A(t) = sqrt(P0 [W]) * sech(t/T0 [ps])

centered at wavelength center_wavelength_nm (nm). time_window (ps) sets temporal grid size.

Optional GDD and TOD are in ps^2 and ps^3.

Note: The full-width-at-half-maximum (FWHM) is given by T0_ps * 1.76

class pynlo.light.DerivedPulses.**GaussianPulse**(*power*, *T0_ps*, *center_wavelength_nm*, *time_window_ps=10.0*, *frep_MHz=100.0*, *NPTS=1024*, *GDD=0*, *TOD=0*, *chirp2=0*, *chirp3=0*, *power_is_avg=False*)

Bases: *pynlo.light.PulseBase.Pulse*

__init__(*power*, *T0_ps*, *center_wavelength_nm*, *time_window_ps=10.0*, *frep_MHz=100.0*, *NPTS=1024*, *GDD=0*, *TOD=0*, *chirp2=0*, *chirp3=0*, *power_is_avg=False*)

Generate Gaussian pulse A(t) = sqrt(peak_power[W]) * exp(-(t/T0 [ps])^2 / 2) centered at wavelength center_wavelength_nm (nm). time_window (ps) sets temporal grid size. Optional GDD and TOD are in ps^2 and ps^3.

Note: For this definition of a Gaussian pulse, T0_ps is the full-width-at-half-maximum (FWHM) of the pulse.

class pynlo.light.DerivedPulses.**FROGPulse**(*time_window_ps*, *center_wavelength_nm*, *power*, *frep_MHz=100.0*, *NPTS=1024*, *power_is_avg=False*, *fileloc=''*, *flip_phase=True*)

Bases: *pynlo.light.PulseBase.Pulse*

__init__(*time_window_ps*, *center_wavelength_nm*, *power*, *frep_MHz=100.0*, *NPTS=1024*, *power_is_avg=False*, *fileloc=''*, *flip_phase=True*)
Generate pulse from FROG data. Grid is centered at wavelength center_wavelength_nm (nm), but pulse properties are loaded from data file. If flip_phase is true, all phase is multiplied by -1 [useful for correcting direction of time ambiguity]. time_window (ps) sets temporal grid size.

power sets the pulse energy: if power_is_epp is True then the number is pulse energy [J] if power_is_epp is False then the power is average power [W], and is multiplied by frep to calculate pulse energy

class pynlo.light.DerivedPulses.**NoisePulse**(*center_wavelength_nm*, *time_window_ps=10.0*, *NPTS=256*, *frep_MHz=None*)

Bases: *pynlo.light.PulseBase.Pulse*

class pynlo.light.DerivedPulses.**CWPulse**(*avg_power*, *center_wavelength_nm*, *time_window_ps=10.0*, *NPTS=256*, *offset_from_center_THz=None*)

Bases: *pynlo.light.PulseBase.Pulse*

gen_OSA(*time_window_ps*, *center_wavelength_nm*, *power*, *power_is_epp=False*, *fileloc='O:\\OFM\\Maser\\Dual-Comb 100 MHz System\\Pump spectrum-Yb-101614.csv'*, *log=True*, *rows=30*)

Generate pulse from OSA data. Grid is centered at wavelength center_wavelength_nm (nm), but pulse properties are loaded from data file. time_window (ps) sets temporal grid size. Switch in place for importing log vs. linear data.

power sets the pulse energy: if power_is_epp is True then the number is pulse energy [J] if power_is_epp is False then the power is average power [W], and is multiplied by frep to calculate pulse energy

3.1.3 pynlo.light.beam

class pynlo.light.beam.**OneDBeam**(*waist_meters=1.0*, *this_pulse=None*, *axis=None*)

Simple Gaussian beam class for propagation and calculating field intensities. Contains beam shape and propagation axis information. The beam waist is held independent of index of refraction, from which the confocal parameter and beam geometry can be calculated.

According to Boyd, who cites Klienman (1966) and Ward and New (1969), it is generally true that the confocal parameter is conserved in harmonic generation and DFG. This parameter is b = 2 pi w0**2 / lambda.

__init__(*waist_meters=1.0*, *this_pulse=None*, *axis=None*)

Initialize class instance. From waist, confocal parameter is derived. A Pulse class is input, and it is assumed that each color focuses to the same waist size at the same point. From this, the (chromatic) confocal parameter b(lambda) is calculated

calc_optimal_beam_overlap_in_crystal(*this_pulse*, *othr_pulse*, *othr_beam*, *crystal_instance*, *L=None*)

Calculate waist w0 for a beam to maximuze the integral (field-square) between it beam and Beam instance second_beam integrated along the length of a crystal. If L is not specified, then the crystal length is used.

calc_overlap_integral(*z*, *this_pulse*, *othr_pulse*, *othr_beam*, *crystal_instance*, *reverse_order=False*)

Calculate overlap integral (field-square) between this beam and Beam instance second_beam inside of a crystal. If reverse_order is true, then the order of second_beam will be reversed.

calculate_R(*z*, *n_s=1.0*)

Calculate beam curvature. : R(z) = z * [1 + (z_R/ z)**2]

calculate_gouy_phase(*z*, *n_s*)

Return the Gouy phase shift due to focusing a distance z in a crystal, where it is assumed that the focus is at crystal_length / 2.0. Return is exp(i psi), as in eq 37 in Siegman Ch 17.4, where A ~ exp(-ikz + i psi).

calculate_waist(*z*, *n_s=1.0*)

Calculate the beam waist a distance z from the focus. The expression is :

w(z) = w0 (1+ (2z/b)**2)**1/2

calculate_zR(*n_s=1.0*)

Calculate Rayleigh range, accounting for index of refraction.

rtP_to_a (*n_s*, *z=None*)
Calculate conversion constant from electric field to average power from pulse and crystal class instances: A ** 2 = rtP_to_a**2 * P

rtP_to_a_2 (*pulse_instance*, *crystal_instance*, *z=None*, *waist=None*)
Calculate conversion constant from electric field to average power from pulse and crystal class instances: A ** 2 = rtP_to_a**2 * P

set_waist_to_match_central_waist (*this_pulse*, *w0_center*, *crystal_instance*)
Calculate waist w0 for a beam match so that all confocal parameters are equal while matching waist w0_center at center color of this beam

set_waist_to_match_confocal (*this_pulse*, *othr_pulse*, *othr_beam*, *crystal_instance*)
Calculate waist w0 for a beam match confocal parameters with othr_beam

3.2 pynlo.interactions

The pynlo.interactions module contains sub-modules to simulate the interaction in both three-wave-mixing (like DFG) and four-wave mixing (like supercontinuum generation).

3.2.1 pynlo.interactions.FourWaveMixing

This module implements the Split-step Fourier Method to solve the Generalized Nonlinear Schrodiner Equation and simulate the propagation of pulses in a Chi-3 nonlinear medium.

class pynlo.interactions.FourWaveMixing.SSFM.**SSFM**(*local_error=0.001*, *dz=1e-05*, *disable_Raman=False*, *disable_self_steepening=False*, *suppress_iteration=True*, *USE_SIMPLE_RAMAN=False*, *f_R=0.18*, *f_R0=0.18*, *tau_1=0.0122*, *tau_2=0.032*)

__init__ (*local_error=0.001*, *dz=1e-05*, *disable_Raman=False*, *disable_self_steepening=False*, *suppress_iteration=True*, *USE_SIMPLE_RAMAN=False*, *f_R=0.18*, *f_R0=0.18*, *tau_1=0.0122*, *tau_2=0.032*)
This initialization function sets up the parameters of the SSFM.

calculate_coherence (*pulse_in*, *fiber*, *num_trials=5*, *random_seed=None*, *noise_type='one_photon_freq'*, *n_steps=50*, *output_power=None*, *reload_fiber_each_step=False*)
This function runs pynlo.interactions.FourWaveMixing.SSFM.propagate() several times (given by num_trials), each time adding random noise to the pulse. By comparing the electric fields of the different pulses, and estimate of the coherence can be made.

The parameters are the same as for pynlo.interactions.FourWaveMixing.SSFM.propagate(), except as listed below

Parameters

- **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()`.

propagate(*pulse_in*, *fiber*, *n_steps*, *output_power=None*, *reload_fiber_each_step=False*)

This is the main user-facing function that allows a pulse to be propagated along a fiber (or other nonlinear medium).

Parameters

- **pulse_in** (*pulse object*) – this is an instance of the *pynlo.light.PulseBase.Pulse* class.
- **fiber** (*fiber object*) – this is an instance of the *pynlo.media.fibers.fiber.FiberInstance* class.
- **n_steps** (*int*) – the number of steps requested in the integrator output. Note: the RK4IP integrator uses an adaptive step size. It should pick the correct step size automatically, so setting n_steps should not affect the accuracy, just the number of points that are returned by this funciton.
- **output_power** – This parameter is a mystery
- **reload_fiber_each_step** (*boolean*) – This flag determines if the fiber parameters should be reloaded every step. It is necessary if the fiber dispersion or gamma changes along the fiber length. *pynlo.media.fibers.fiber.FiberInstance.set_dispersion_function()* and *pynlo.media.fibers.fiber.FiberInstance.set_dispersion_function()* should be used to specify how the dispersion and gamma change with the fiber length

Returns

- **z_positions** (*array of float*) – an array of z-positions along the fiber (in meters)
- **AW** (*2D array of complex128*) – A 2D numpy array corresponding to the intensities in each *frequency* bin for each step in the z-direction of the fiber.
- **AT** (*2D array of complex128*) – A 2D numpy array corresponding to the intensities in each *time* bin for each step in the z-direction of the fiber.
- **pulse_out** (*PulseBase object*) – the pulse after it has propagated through the fiber. This object is suitable for propagation through the next fiber!

propagate_to_gain_goal(*pulse_in*, *fiber*, *n_steps*, *power_goal=1*, *scalefactor_guess=None*, *powertol=0.05*)

Integrate over length of gain fiber such that the average output power is power_goal [W]. For this to work, fiber must have spectroscopic gain data from an amplifier model or measurement. If the approximate scalefactor needed to adjust the gain is known it can be passed as scalefactor_guess.

This function returns a tuple of tuples:

((ys,AWs,ATs,pulse_out), scale_factor)

3.2.2 pynlo.interactions.ThreeWaveMixing

This module simulated DFG in a Chi-2 medium.

class pynlo.interactions.ThreeWaveMixing.DFG_integrand.**dfg_problem**(*pump_in*, *sgnl_in*, *crystal_in*, *disable_SPM=False*, *pump_waist=1e-05*, *apply_gouy_phase=False*, *plot_beam_overlaps=False*, *wg_mode=False*, *Aeff_squm=None*)

This class defines the integrand for a DFG or OPO parametric inteaction. Following Eqn (8) in Seres & Hebling, "Nonstationary theory of synchronously pumped femtosecond optical parametric oscillators", JOSA B Vol 17 No 5, 2000.

Ai(*y*)

Ap(*y*)

As(*y*)

deriv(*z*, *y*, *dydx*)

format_overlap_plots()

gen_jl(*y*)

Following Eqn (8) in Seres & Hebling, "Nonstationary theory of synchronously pumped femtosecond optical parametric oscillators", JOSA B Vol 17 No 5, 2000. A call to this function updates the :math: *chi_3* mixing terms used for four-wave mixing.

Parameters **y** (*array-like, shape is 3 * NPTS*) – Concatenated pump, signal, and idler fields

helper_dxdy(*x*, *y*)

idlr_P_to_a = None

last_calc_z = -1000000.0

overlap_idlr = None

overlap_pump = None

overlap_sgnl = None

poling(*x*)

Helper function to get sign of :math: *d_ extrm{eff}* at position :math: *x* in the crystal. Uses self.crystal's pp function.

For APPLN this is somewhat complicated. The input position x could be many periods away from the previous value, and in either direction. One solution would be carefully stepping back and forth, but this needs to be perfect to prevent numerical errors.

Instead, precompute the domain boundaries and use a big comparison to check the poling(z)

Returns **x** – Sign (+1 or -1) of :math: *d_ extrm{eff}*.

Return type int

precompute_poling()

process_stepper_output (*solver_out*)
Post-process output of ODE solver.

The saved data from an ODE solved are the pump, signal, and idler in the dispersionless reference frame. To see the pulses "as they really are", this dispersion must be added back in.

Parameters **solver_out** – Output class instance from ODESolve

Returns Instance of dfg_results_interface class

Return type dfg_results

pump_P_to_a = None

sgnl_P_to_a = None

vg (*n*, *wl*)

3.3 pynlo.media

The **media** module contains sub-modules for modeling fibers and crystals.

3.3.1 pynlo.media.fibers

These classes are used to model fibers or fiber-like waveguides.

class pynlo.media.fibers.fiber.**FiberInstance** (*fiber_db='general_fibers'*, *fiber_db_dir=None*)
This is a class that contains the information about a fiber.

Beta2 (*pulse*)
This provides the beta_2 (in ps^2 / meter).

Beta2_to_D (*pulse*)
This provides the dispersion parameter D (in ps / nm / km) at each frequency of the supplied pulse

betas = None

fiberspecs = {}

fibertype = None

gamma = None

generate_fiber (*length*, *center_wl_nm*, *betas*, *gamma_W_m*, *gain=0*, *gvd_units='ps^n/m'*, *label='Simple Fiber'*)
This generates a fiber instance using the beta-coefficients.

get_betas (*pulse*, *z=0*)
This provides the propagation constant (beta) at the frequencies of the supplied pulse grid. The units are 1/meters.

Two 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. However, since this expansion is done without the lower order coefficients, the first two terms of the Taylor expansion are not defined. In

order to provide a nice input for the SSFM, which assumes that the group velocity will be zero at the pulse central frequency, the slope and offset at the pump central frequency are set to zero.

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** (an instance of the `pynlo.light.pulse.PulseBase` class) – the pulse must be supplied in order for the frequency grid to be known

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_gain(*pulse*, *output_power=1*)
Retrieve gain spectrum for fiber. If fiber has 'simple gain', this is a scalar. If the fiber has a gain spectrum (eg EDF or YDF), this will return this spectrum as a vector corresponding to the Pulse class frequency axis. In this second case, the output power must be specified, from which the gain/length is calculated.

get_gamma(*z=0*)
Allows the gamma (effective nonlinearity) to be queried 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

length = None

load_dispersion()
This is typically called by the "load_from_db" function. It takes the values from the self.fiberspecs dict and transfers them into the appropriate variables.

load_from_db(*length*, *fibertype*, *poly_order=2*)
This loads a fiber from the database.

load_from_file(*filename*, *length=0.1*, *fiberName=None*, *gamma_W_m=0*, *gain=0*, *alpha=0*, *delimiter=', '*, *skiprows=0*, *poly_order=3*)
This loads dispersion give the path of a file. The file is expected to be in the format wavelength (nm), D (ps/nm/km).

poly_order = None

set_dispersion_function(*dispersion_function*, *dispersion_format='GVD'*)
This allows the user to provide a function for the fiber dispersion that can vary as a function of *z*, the length along the fiber. 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*) – returning D or Beta coefficients as a function of z
- **dispersion_formats** (*'GVD' or 'D' or 'n'*) – determines if the dispersion will be identified in terms of Beta coefficients (GVD, in units of ps^2/m, not ps^2/km) or D (ps/nm/km) n (effective refractive index)

Notes

For example, this code will create a fiber where Beta2 changes from anomalous to zero along the fiber:

```
Length = 1.5

def myDispersion(z):

    frac = 1 - z/(Length)

    beta2 = frac * -50e-3
    beta3 = 0
    beta4 = 1e-7

    return beta2, beta3, beta4
```

fiber1 = fiber.FiberInstance() fiber1.generate_fiber(Length, center_wl_nm=800, betas=myDispersion(0), gamma_W_m=1)

fiber.set_dispersion_function(myDisperion, dispersion_format='GVD')

set_gamma_function(*gamma_function*)

This allows the user to provide a function for 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*) – returning gamma function of z

Created on Tue Jan 28 13:56:17 2014 This file is part of pyNLO.

pyNLO is free software: you can redistribute it and/or modify it under the terms of the GNU General Public gLicense as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

pyNLO is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with pyNLO. If not, see <http://www.gnu.org/licenses/>.

@author: dim1

pynlo.media.fibers.calculators.**DTabulationToBetas**(*lambda0*, *DData*, *polyOrder*, *DDataIsFile=True*, *return_diagnostics=False*)

Read in a tabulation of D vs Lambda. Returns betas in array [beta2, beta3, ...]. If return_diagnostics is True, then return (betas, fit_x_axis (omega in THz), data (ps^2), fit (ps^2))

3.3.2 pynlo.crystals

These classes are used to model various nonlinear crystals.

class pynlo.media.crystals.CrystalContainer.**Crystal**(*params*)

Container for chi-2 nonlinear crystals. Actual crystal refractive index, dispersion, and nonlinearity information is stored in modular files. Read these in by calling <crystal>.load(crystal_instance, params).

calculate_D_fs_um_mm(*wavelengths_nm*, *axis=None*)

Calculate crystal dispersion at 'wavelengths_nm' along 'axis' in short crystal, broad bandwidth units of fs/um/mm

calculate_D_ps_nm_km(*wavelengths_nm*, *axis=None*)

Calculate crystal dispersion at 'wavelengths_nm' [nm] along 'axis' in standard photonic engineering units ps/nm/km

calculate_group_velocity_nm_ps (*wavelengths_nm*, *axis=None*)
Calculate group velocity vg at 'wavelengths_nm' [nm] along 'axis' in units of nm/ps

calculate_mix_phasematching_bw (*pump_wl_nm*, *signal_wl_nm*, *axis=None*)

Calculate the phase matching bandwidth in the case of mixing between narrowband pump (highest photon energy) with a signal field. The bandwidths of mixing between pump-signal and pump-idler are calculated, and the smaller of the two is returned.

Parameters

- **pump_wl_nm** (*float*) – Wavelength of pump field, bandwidth assumed to be 0 [nm]
- **signal_wl_nm** (*array-like*) – Wavelength of signal field [nm]

Returns **acceptance bandwidth** – Phasematching bandwidth [m^-1 * m]

Return type float

References

Peter E Powers, "Fundamentals of Nonlinear Optics", pp 106

calculate_pulse_delay_ps (*wl1_nm*, *wl2_nm*, *crystal_length_mks=None*, *axis=None*)
Calculate the pulse delay between pulses at wl1 and wl2 after crystal. Be default, crystal instance's length is used.

get_pulse_k (*pulse_instance*, *axis=None*)
Return vector of angular wavenumbers (m^-1) for the pulse_instance's frequency grid inside the crystal

get_pulse_n (*pulse_instance*, *axis=None*)
Return vector of indices of refraction for the pulse_instance's frequency grid inside the crystal

invert_dfg_qpm_to_signal_wl (*pump_wl_nm*, *poling_period_mks*, *max_signal_wl_nm=2000*)

Calculate the signal wavelength phasematched in QPM by the given poing period for the specified pump wavelength.

Parameters

- **pump_wl_nm** (*float*) – Wavelength of pump field, bandwidth assumed to be 0 [nm]
- **poling_period_mks** (*float*) – Period length of the QPM grating

Returns **Signal wavelength [nm]**

Return type float

set_caching (*cache_enable=True*)
Enable or disable caching of refractive indices. Enabling this uses more memory, but can save costly recomputations

Parameters **cache_enable** (*bool*) –

. . . ..More undocumented crystals here. . . .

3.4 pynlo.util.ode_solve

These classes are an adaptation of the very nice *Numerical Recipes* ODE solvers into Python. The solver is divided into two parts: specific step iterators (eg Dopri853) and the framework for stepping through the ODE (steppers)

3.4.1 Dormand-Prince 853 Stepper

class pynlo.util.ode_solve.dopr853.**StepperDopr853**(*yy*, *dydxx*, *xx*, *atoll*, *rtoll*, *dens*)
Bases: *pynlo.util.ode_solve.steppers.StepperBase*

3.4.2 Steppers and helpers

class pynlo.util.ode_solve.steppers.**Output**(*nsaves=None*)
The output class is used by the ode solver to store the integrated output at specified *x* values. In addition to housing the matrices containing the *x* and *y* data, the class also provides a simple function call to store new data and resizes the output grids dynamically.

Parameters **nsaves** – Number of anticipated save points, used for calculating value of *x* at which integrand will be evaluted and saved.

init(*neqn*, *xlo*, *xhi*, *dtype=<type 'numpy.float64'>*)
Setup routine, which creates the output arrays. If nsaves was provided at class initialization, the positions at which the integrand will be saved are also calculated.

Parameters

- **neqn** – Number of equations, or the number of y values at each x.
- **xlo** – Lower bound of integration (start point.)
- **xhi** – Upper bound of integration (stop point.)
- **dtype** – Data type of each y. Any Python data type is acceptable.

out(*nstp*, *x*, *y*, *s*, *h*)
nstp is current step number, current values are x & y, Stepper is s and step size is h

class pynlo.util.ode_solve.steppers.**StepperBase**(*yy*, *dydxx*, *xx*, *atoll*, *rtoll*, *dense*)

class pynlo.util.ode_solve.steppers.**ODEint**(*ystartt*, *xx1*, *xx2*, *atol*, *rtol*, *h1*, *hminn*, *outt*, *stepper_class*, *RHS_class*, *dense=True*, *dtype=None*)

__init__(*ystartt*, *xx1*, *xx2*, *atol*, *rtol*, *h1*, *hminn*, *outt*, *stepper_class*, *RHS_class*, *dense=True*, *dtype=None*)
Class for integrating ODEs.

Notes

This code is based upon *Numerical Recipes 3rd edition*'s imlementation, but with some changes due to the translation: 1.) The ODE is passed as a class instance 'RHS_class'. This class must

have a member function deriv(x,y,dydx) which calculates the RHS and writes the value into dydx.

2.) Unlike the NR version, ODEint is not derived from the stepper. instead, the stepper class to be used is passed to the ODEint constructor (stepper_class).

3.) As a consequence of (2), x and y are stored in the stepper instance (ODEint.s) and not in ODEint iteself.

3.5 pynlo.devices

class pynlo.devices.grating_compressor.**TreacyCompressor**(*lines_per_mm*, *incident_angle_degrees*)

This class calculates the effects of a grating-based pulse compressor, as described in E. B. Treacy, "Optical Pulse Compression With Diffraction Gratings", IEEE Journal of Quantum Electronics QE5(9), p454 (1969): http://dx.doi.org/10.1109/JQE.1969.1076303

It implements eqn 5b from Treacy1969:

```
-4 pi^2 c b
```

{1} dt/dw = ——————————————— w^3 d^2 (1- (2 pi c/ wd - sin gamma)^2)

where gamma is the diffraction angle, w is the angular frequency, d is the grating ruling period, and b is the slant distance between gratings,

```
{1b} b = G sec(gamma - theta)
```

where G is the grating separation and theta is the acute angle between indicent and diffracted rays (text before eq 4). The grating equation :: relates the angles (generalized eq 3):

{2} sin(gamma - theta) + sin(gamma) = m lambda / d

More conventionally, the grating equation is cast in terms of the incident and diffracted ray angles,

```
{3} sin(alpha) + sin(beta) = m lambda / d.
```

It makes sense to solve {3} using the grating specifications (eg for optimum incident angle a) and then derive Treacy's theta and gamma:

```
{4} gamma = alpha                    theta = gamma - alpha
```

This code only considers first order diffraction, as most gratings are designed for this (eg LightSmyth transmission gratings.)

apply_phase_to_pulse(*grating_separation_meters*, *pulse*)

Apply grating disersion (all orders) to a Pulse instance. Phase is computed by numerical integration of dphi/domega (from Treacy)

calc_compressor_HOD(*wavelength_nm*, *grating_separation_meters*, *dispersion_order*)

Calculate higher order dispersion by taking w - derivatives of dt/dw

calc_compressor_dnphi_domega_n(*wavelength_nm*, *grating_separation_meters*, *dispersion_order*)

Calculate higher order dispersion by taking w - derivatives of dt/dw

CHAPTER 4

Examples

4.1 Supercontinuum generation example

Here is an example of supercontinuum generation in a fiber

```
import numpy as np
import matplotlib.pyplot as plt
import pynlo

FWHM    = 0.050  # pulse duration (ps)
pulseWL = 1550   # pulse central wavelength (nm)
EPP     = 50e-12 # Energy per pulse (J)
GDD     = 0.0    # Group delay dispersion (ps^2)
TOD     = 0.0    # Third order dispersion (ps^3)

Window  = 10.0   # simulation window (ps)
Steps   = 50     # simulation steps
Points  = 2**13  # simulation points

beta2   = -120     # (ps^2/km)
beta3   = 0.00     # (ps^3/km)
beta4   = 0.005    # (ps^4/km)

Length  = 20    # length in mm

Alpha   = 0.0     # attentuation coefficient (dB/cm)
Gamma   = 1000    # Gamma (1/(W km)

fibWL   = pulseWL # Center WL of fiber (nm)

Raman   = True    # Enable Raman effect?
Steep   = True    # Enable self steepening?

alpha = np.log((10**(Alpha * 0.1))) * 100  # convert from dB/cm to 1/m
```

(continues on next page)

(continued from previous page)

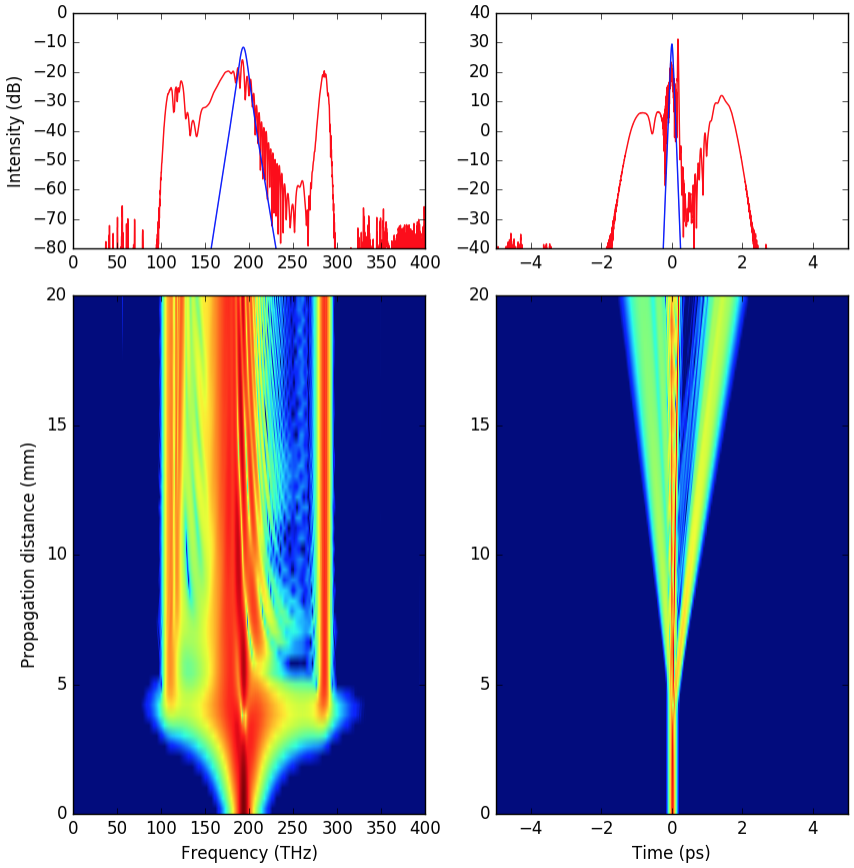
```
# set up plots for the results:
fig = plt.figure(figsize=(10,10))
ax0 = plt.subplot2grid((3,2), (0, 0), rowspan=1)
ax1 = plt.subplot2grid((3,2), (0, 1), rowspan=1)
ax2 = plt.subplot2grid((3,2), (1, 0), rowspan=2, sharex=ax0)
ax3 = plt.subplot2grid((3,2), (1, 1), rowspan=2, sharex=ax1)


######## This is where the PyNLO magic happens! ############################

# create the pulse!
pulse = pynlo.light.DerivedPulses.SechPulse(1, FWHM/1.76, pulseWL, time_window_
↪ps=Window,
                    GDD=GDD, TOD=TOD, NPTS=Points, frep_MHz=100, power_is_avg=False)
pulse.set_epp(EPP) # set the pulse energy

# create the fiber!
fiber1 = pynlo.media.fibers.fiber.FiberInstance()
fiber1.generate_fiber(Length * 1e-3, center_wl_nm=fibWL, betas=(beta2, beta3, beta4),
                              gamma_W_m=Gamma * 1e-3, gvd_units='ps^n/km', gain=-
↪alpha)

# Propagation
evol = pynlo.interactions.FourWaveMixing.SSFM.SSFM(local_error=0.001, USE_SIMPLE_
↪RAMAN=True,
                disable_Raman=np.logical_not(Raman),
                disable_self_steepening=np.logical_not(Steep))

y, AW, AT, pulse_out = evol.propagate(pulse_in=pulse, fiber=fiber1, n_steps=Steps)

########## That's it! Physic done. Just boring plots from here! ###############


F = pulse.W_mks / (2 * np.pi) * 1e-12 # convert to THz

def dB(num):
    return 10 * np.log10(np.abs(num)**2)

zW = dB( np.transpose(AW)[:, (F > 0)] )
zT = dB( np.transpose(AT) )

y = y * 1e3 # convert distance to mm


ax0.plot(F[F > 0],  zW[-1], color='r')
ax1.plot(pulse.T_ps,zT[-1], color='r')

ax0.plot(F[F > 0],   zW[0], color='b')
ax1.plot(pulse.T_ps, zT[0], color='b')


extent = (np.min(F[F > 0]), np.max(F[F > 0]), 0, Length)
ax2.imshow(zW, extent=extent, vmin=np.max(zW) - 60.0,
                vmax=np.max(zW), aspect='auto', origin='lower')
```

(continues on next page)

(continued from previous page)

```
extent = (np.min(pulse.T_ps), np.max(pulse.T_ps), np.min(y), Length)
ax3.imshow(zT, extent=extent, vmin=np.max(zT) - 60.0,
           vmax=np.max(zT), aspect='auto', origin='lower')


ax0.set_ylabel('Intensity (dB)')

ax2.set_xlabel('Frequency (THz)')
ax3.set_xlabel('Time (ps)')

ax2.set_ylabel('Propagation distance (mm)')

ax2.set_xlim(0,400)

ax0.set_ylim(-80,0)
ax1.set_ylim(-40,40)

plt.show()
```

Output:



CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

A

B

C

P

R

S

T

V

W