PyNLO Documentation

Release 0.1

Gabriel Ycas

March 17, 2016

Contents

Contents:

Welcome to pyNLO's documentation!

1.1 Package Design

Information about the design of the package and its classes.

1.1.1 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.

1.1.2 Pulse Class

class `pynlo.light.`**`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

`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_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

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

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]

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

1.1.3 DFG Integrand

Difference frequency generation module

Defines:

- The dfg_problem, a class which can be intergrated by the pyNLO ODESolve
- The fftcomputer, which handles FFTs using pyFFTW
- A helper class, dfg_results_interface, which provides a Pulse-class based wrapper around the dfg results.

Authors: Dan Maser, Gabe Ycas

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*)

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.

`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

`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.

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

Return type int

`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

class `pynlo.interactions.ThreeWaveMixing.DFG_integrand.`**`dfg_results_interface`**(*integrand_instance*, *pump*, *sgnl*, *idlr*, *z*)

Interface to output of DFG solver. This class provides a clean way of working with the DFG field using the Pulse class.

Notes

After initialization, calling:

```
get_{pump,sgnl,idlr}(n)
```

will set the dfg results class' "pulse" instance to the appropriate field and return it.

Example

To plot the 10th saved signal field, you would call:

```
p = dfg_results_interface.get_sgnl(10-1)
plt.plot(p.T_ps, abs(p.AT)**2 )
```

To get the actual position (z [meters]) that this corresponds to, call:

```
z = dfg_results_interface.get_z(10-1)
```

1.1.4 Numerical Recipes-based ODESolve

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)

1.1.5 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*)

1.1.6 Dormand-Prince 853 Stepper

class `pynlo.util.ode_solve.dopr853.`**`StepperDopr853`**(*yy*, *dydxx*, *xx*, *atoll*, *rtoll*, *dens*)

Bases: `pynlo.util.ode_solve.steppers.StepperBase`

1.1.7 Fiber Class

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

p

A

C

D

F

G

I

O

P

R

S

T

V

W