# pypret Documentation

*Release 0.1alpha*

**Nils C. Geib**

**Jul 03, 2019**

# Contents

**Release** 0.1alpha

**Date** Jul 03, 2019

This is the documentation of *Python for pulse retrieval*. It is a Python package that aims to provide algorithms and tools to retrieve ultrashort laser pulses from parametrized nonlinear process spectra, such as frequency-resolved optical gating (FROG), dispersion scan (d-scan), time-domain ptychography (TDP) or multiphoton intrapulse interference phase scan (MIIPS).

The package is currently in an early alpha state. It provides the algorithms but still requires thorough understanding of what they do to apply them correctly on measured data.

# CHAPTER 1

# Background

The package was developed at the Institute of Applied Physics at the Friedrich Schiller University Jena. Main author is Nils C. Geib. You can reach me at nils.geib@uni-jena.de if you have questions or comments on the code.

The current capabilities of the package reflect mostly what we presented in our publication on a common pulse retrieval algorithm [Geib2019]. If you want to reference this package you may cite that paper.

The code in its current state mainly serves to give a reference implementation of the algorithms discussed within and allow the reproduction of our results. It is planned, however, to expand the package to make it a more full-fledged solution for pulse retrieval.

# CHAPTER 2

# User documentation

## 2.1 Installation

Installation with `pip` or `conda` is currently neither supported nor necessary. Just clone the code repository from git:

```
git clone https://github.com/ncgeib/pypret.git
```

and the directory `pypret` within contains all the required code of the package. Either add its location to your PYTHONPATH or copy it in your working directory.

As the package matures I may add an installer.

### 2.1.1 Requirements

It requires Python >=3.6 and recent versions of NumPy and SciPy. Furthermore, it requires `h5py` for storage and loading. Optional dependencies are

- pyfftw (for faster FFTs)
- numba (for optimization of some low-level routines)
- python-magic (to recognize zipped HDF5 files)

## 2.2 Getting started

pypret is a package to simulate and retrieve from measurements such as frequency-resolved optical gating (FROG), dispersion scan (d-scan), interferometric FROG (iFROG), time-domain ptychography (TDP) and even multiphoton intrapulse interference phase scan (MIIPS). These are all measurements used for ultrashort (sub-ps) laser pulse measurement. More generally the package can handle all kinds of parametrized nonlinear process spectra (PNPS) measurements.

A good place to start reading on the algorithms and the used notation is our paper [Geib2019] and its supplement. pypret can be thought to accompany this publication and can be used to reproduce most of the results shown there.

### 2.2.1 Basic Use

pypret can be used to simulate PNPS measurements. This is useful for designing experiments and necessary for retrieval, of course.

In a first step you have to set up the simulation grid in time and frequency:

```
ft = pypret.FourierTransform(256, dt=2.5e-15)
```

which generates a 256 elements grid with a temporal spacing of 2.5 fs centered around t=0. The frequency grid is chosen to match the reciprocity relation `dt * dw = 2 * pi / N`. Alternatively you can specify the frequency spacing. See the documentation at *pypret.fourier module*. Next you can instantiate a `pypret.Pulse` object:

```
pulse = pypret.Pulse(ft, 800e-9)
```

where we used a central wavelength of 800 nm. This class can already be used for small but useful calculations:

```
# generate pulse with Gaussian spectrum and field standard deviation
# of 20 nm
pulse.spectrum = pypret.lib.gaussian(pulse.wl, x0=800e-9, sigma=20e-9)
# print the accurate FWHM of the temporal intensity envelope
print(pulse.fwhm(dt=pulse.dt/100))
# propagate it through 1cm of BK7 (remove first ord)
phase = np.exp(1.0j * pypret.material.BK7.k(pulse.wl) * 0.01)
pulse.spectrum = pulse.spectrum * phase
# print the temporal FWHM again
print(pulse.fwhm(dt=pulse.dt/100))
# finally plot the pulse
pypret.graphics.PulsePlot(pulse)
```

You can now instantiate a PNPS class with that pulse object:

```
insertion = np.linspace(-0.025, 0.025, 128)  # insertion in m
pnps = pypret.PNPS(pulse, "dscan", "shg", material=pypret.material.BK7)
# calculate the measurement trace
pnps.calculate(pulse.spectrum, delay)
original_spectrum = pulse.spectrum
# and plot it
pypret.MeshDataPlot(pnps.trace)
```

The PNPS constructor supports a lot of different PNPS measurements (see docs at *pypret.pnps module*). Furthermore, it is easy to implement your own.

Finally, you can use pypret for pulse retrieval by instantiating a Retriever object:

```
# do the retrieval
ret = pypret.Retriever(pnps, "copra", verbose=True, maxiter=300)
# start with a Gaussian spectrum with random phase as initial guess
pypret.random_gaussian(pulse, 50e-15, phase_max=0.0)
# now retrieve from the synthetic trace simulated above
ret.retrieve(pnps.trace, pulse.spectrum)
# and print the retrieval results
ret.result(original_spectrum)
```

A lot of different retrieval algorithms besides the default, COPRA, are implemented (see docs at *pypret.retrieval package*). While COPRA should work for all PNPS measurements, you may try one of the others for verification.

### 2.2.2 Storage

The *pypret.io package* subpackage supports saving almost arbitrary Python structures and all pypret classes to HDF5 files. You can either use the `pypret.save()` function or the *save* method on classes:

```
pnps.calculate(pulse.spectrum, insertion)
pnps.trace.save("trace.hdf5")
# or
pypret.save(pnps.trace, "trace.hdf5")
# load it with
trace = pypret.load("trace.hdf5")
```

This should make storing intermediate or final results almost effortless.

### 2.2.3 Experimental data

As this question is surely going to come: you can use pypret to retrieve pulses from experimental data, however, it currently has no pre-processing functions to make that convenient. The data fed to the retrieval functions has to be properly dark-subtracted and interpolated. Furthermore, some features that are very useful for retrieval from experimental data (e.g., handling non-calibrated traces) are not yet implemented. This is on the top of the ToDo-list, though.

## 2.3 References

# CHAPTER 3

# API documentation

## 3.1 pypret.fourier module

This module implements the Fourier transforms on linear grids.

The following code approximates the continuous Fourier transform (FT) on equidistantly spaced grids. While this is usually associated with 'just doing a fast Fourier transform (FFT)', surprisingly, much can be done wrong.

The reason is that the correct expressions depend on the grid location. In fact, the FT can be calculated with one DFT but in general it requires a prior and posterior multiplication with phase factors.

The FT convention we are going to use here is the following:

```
(w) = 1/2pi  E(t) exp(+i w t) dt
E(t) =        (w) exp(-i t w) dw
```

where w is the angular frequency. We can approximate these integrals by their Riemann sums on the following equidistantly spaced grids:

```
t_k = t_0 + k Δt, k=0, ..., N-1
w_n = w_0 + n Δw, n=0, ..., N-1
```

and define E_k = E(t_k) and _n = (w_n) to obtain:

```
_n = Δt/2pi _k E_k exp(+i w_n t_k)
E_k = Δw     _n _n exp(-i t_k w_n).
```

To evaluate the sum using the FFT we can expand the exponential to obtain:

```
_n = Δt/2pi exp(+i n t_0 Δw) _k [E_k exp(+i t_k w_0) ] exp(+i n k Δt Δw)
E_k = Δw     exp(-i t_k w_0)  _n [_n exp(-i n t_0 Δw)] exp(-i k n Δt Δw)
```

Additionally, we have to require the so-called reciprocity relation for the grid spacings:

```
      !
Δt Δw = 2pi / N = ζ      (reciprocity relation)
```

This is what enables us to use the DFT/FFT! Now we look at the definition of the FFT in NumPy:

```
 fft[x_m] -> X_k =     _m exp(-2pi i m k / N)
ifft[X_k] -> x_m = 1/N _k exp(+2pi i k m / N)
```

which gives the final expressions:

```
_n = Δt N/2pi r_n   ifft[E_k s_k  ]
E_k = Δw       s_k^*   fft[_n r_n^*]

with r_n = exp(+i n t_0 Δw)
     s_k = exp(+i t_k w_0)
```

where ^* means complex conjugation. We see that the array to be transformed has to be multiplied with an appropriate phase factor before and after performing the DFT. And those phase factors mainly depend on the starting points of the grids: w_0 and t_0. Note also that due to our sign convention for the FT we have to use ifft for the forward transform and vice versa.

Trivially, we can see that for `w_0 = t_0 = 0` the phase factors vanish and the FT is approximated well by just the DFT. However, in optics these grids are unusual. For `w_0 = l Δw` and `t_0 = m Δt`, where l, m are integers (i.e., w_0 and t_0 are multiples of the grid spacing), the phase factors can be incorperated into the DFT. Then the phase factors can be replaced by circular shifts of the input and output arrays.

This is exactly what the functions (i)fftshift are doing for one specific choice of l and m, namely for:

```
t_0 = -floor(N/2) Δt
w_0 = -floor(N/2) Δw.
```

In this specific case only we can approximate the FT by:

```
_n = Δt N/2pi fftshift(ifft(ifftshift(E_k)))
E_k = Δw       fftshift( fft(ifftshift(_n)))  (no mistake!)
```

We see that the ifftshift _always_ has to appear on the inside. Failure to do so will still be correct for even N (here fftshift is the same as ifftshift) but will produce wrong results for odd N.

Additionally you have to watch out not to violate the assumptions for the grid positions. Using a symmetrical grid, e.g.,:

```
x = linspace(-1, 1, 128)
```

will also produce wrong results, as the elements of x are not multiples of the grid spacing (but shifted by half a grid point).

The main drawback of this approach is that circular shifts are usually far more time- and memory-consuming than an elementwise multiplication, especially for higher dimensions. In fact I see no advantage in using the shift approach at all. But for some reason it got stuck in the minds of people and you find the notion of having to re-order the output of the DFT everywhere.

Long story short: here we are going to stick with multiplying the correct phase factors. The code tries to follow the notation used above.

Good, more comprehensive expositions of the issues above can be found in [Briggs1995] and [Hansen2014]. For the reason why the first-order approximation to the Riemann integral suffices, see [Trefethen2014].

**class** `pypret.fourier.`**`FourierTransform`**(*N*, *dt=None*, *dw=None*, *t0=None*, *w0=None*)

**`backward`**(*x*, *out=None*)

Calculates the backward (inverse) Fourier transform of `x`.

For n-dimensional arrays it operates on the last axis, which has to match the size of *x*.

**Parameters**

- **x** (*`ndarray`*) – The array of which the Fourier transform will be calculated.
- **out** (*`ndarray or None, optional`*) – A location into which the result is stored. If not provided or None, a freshly-allocated array is returned.

**`forward`**(*x*, *out=None*)

Calculates the (forward) Fourier transform of `x`.

For n-dimensional arrays it operates on the last axis, which has to match the size of *x*.

**Parameters**

- **x** (*`ndarray`*) – The array of which the Fourier transform will be calculated.
- **out** (*`ndarray or None, optional`*) – A location into which the result is stored. If not provided or None, a freshly-allocated array is returned.

## 3.2 pypret.pulse module

Provides a class to simulate an ultrashort optical pulse using its envelope description.

The temporal envelope is denoted as *field* and the spectral envelope as *spectrum* in the code and the function signatures.

**class** `pypret.pulse.`**`Pulse`**(*ft*, *wl0*, *unit='wl'*)

Bases: `pypret.io.io.IO`

A class for modelling femtosecond pulses by their envelope.

**`__init__`**(*ft*, *wl0*, *unit='wl'*)

Initializes an optical pulse described by its envelope.

**Parameters**

- **ft** (`FourierTransform`) – A `FourierTransform` instance that specifies a temporal and spectral grid.
- **wl0** (*`float`*) – The center frequency of the pulse.
- **unit** (*`str`*) – The unit in which the center frequency is specified. Can be either of `wl`, `om`, `f`, or `k`. See `frequencies` for more information. Default is `wl`.

**`amplitude`**

The temporal amplitude profile of the pulse in vacuum.

Only read access.

**`copy`**()

Returns a copy of the pulse object.

Note that they still reference the same *FourierTransform* instance, which is assumed to be immutable.

**`field`**
: The complex-valued temporal envelope of the pulse.

  On read access returns a copy of the internal array. On write access the spectral envelope is automatically updated.

**`field_at`**(*t*)
: The complex-valued temporal envelope of the pulse at the times *t*.

**`fwhm`**(*dt=None*)
: Calculates the full width at half maximum (FWHM) of the temporal intensity profile.

  **Parameters** **dt** (*float or None, optional*) – Specifies the required accuracy of the calculation. If *None* (the default) it is only as good as the spacing of the underlying simulation grid - which can be quite coarse compared to the FWHM. If smaller it is calculated based on trigonometric interpolation.

**`intensity`**
: The temporal intensity profile of the pulse in vacuum.

  Only read access.

**`phase`**
: The temporal phase of the pulse.

  Only read access.

**`spectral_amplitude`**
: The spectral amplitude profile of the pulse in vacuum.

  Only read access.

**`spectral_intensity`**
: The spectral intensity profile of the pulse in vacuum.

  Only read access.

**`spectral_phase`**
: The spectral phase of the pulse.

  Only read access.

**`spectrum`**
: The complex-valued spectral envelope of the pulse.

  On read access returns a copy of the internal array. On write access the temporal envelope is automatically updated.

**`spectrum_at`**(*w*)
: The complex-valued spectral envelope of the pulse at the frequencies *w*.

**`time_bandwidth_product`**
: Calculates the rms time-bandwidth product of the pulse.

  In this definition a transform-limited Gaussian pulse has a time-bandwidth product of 0.5. So the number returned by this function will always be >= 0.5.

**`update_field`**()
: Manually updates the field from the (modified) spectrum.

**`update_spectrum`**()
: Manually updates the spectrum from the (modified) field.

`pypret.random_pulse.`**`random_pulse`**(*pulse*, *tbp*, *edge_value=None*, *check=True*)
: Creates a random pulse with a specified time-bandwidth product.

**Parameters**

- **pulse** (*Pulse instance*) –
- **tbp** (*float*) – The specified time-bandwidth product.
- **edge_value** (*float, optional*) – The maximal value for the pulse amplitude at the edges of the grid. It defaults to the double value epsilon ~2e-16.

**Returns** **bool** – is stored in the Pulse instance passed to the function.

**Return type** True on success, False if an error occured. The resulting pulse

### Notes

The function creates random pulses by iteratively restricting the bandwidth in time and frequency domain. It starts from random complex values in frequency domain, multiplies a Gaussian function, transforms in the time domain and multiplies a Gaussian function again. The filter functions are Gaussians with the specified time-bandwidth product. The TBP of the Gaussian filters, however, does not directly correspond to the TBP of the resulting pulse. To use this algorithm to generate a pulse with exactly the specified TBP, it is run in the range 0.5 * TBP to 1.5 * TBP using a scalar root search (brentq). Usually this guarantees convergence within a few tries. The larger the TBP the larger the number of points has to be. So the algorithm may fail to find a solution if pulse.N is too small.

`pypret.random_pulse.`**`random_gaussian`**(*pulse*, *fwhm*, *phase_max=0.3141592653589793*)
Generates a Gaussian pulse with random phase.

Its pulse of duration is given by `fwhm`.

## 3.3 pypret.pnps module

This module provides classes to calculate parametrized nonlinear process spectra (PNPS), such as frequency-resolved optical gating (FROG), interferometric FROG (iFROG), dispersion scan (d-scan), time-domain ptychography (TDP) and pulse-shaper assisted methods such as multiphoton intrapulse interference phase scan (MIIPS).

The code follows the notation used in [Geib2019] and its supplement.

Currently only the abovementioned methods are implemented. But the code is written in such way that including new pulse measurement methods is very easy. If it is a method using a collinear nonlinearity, subclass from *CollinearPNPS*, otherwise from *NoncollinearPNPS*.

In the collinear case only *self.mask(parameter)* has to be implemented which calculates the used linear parametrization operator. In the non-collinear case the function *_calculate* has to be implemented which calculates and returns the PNPS trace `T_mn` and the PNPS signal `S_mk`.

### 3.3.1 Public interface

`pypret.pnps.`**`PNPS`**(*pulse: pypret.pulse.Pulse*, *method: str*, *process: str*, ***kwargs*) → pypret.pnps.BasePNPS
Creates a PNPS instance.

**Parameters**

- **pulse** (`Pulse`) – A pulse instance that is used to simulate the PNPS trace.
- **method** (*str*) –

  **The type of PNPS measurement. Should be one of**

  - 'frog' *(see here)*
  - 'tdp' *(see here)*
  - 'dscan' *(see here)*
  - 'miips' *(see here)*
  - 'ifrog' *(see here)*

- **process** (*str*) –

  **The nonlinear process used in the measurement method. Can be one of**

  - 'shg' : second harmonic generation
  - 'thg' : third harmonic generation
  - 'sd' : self-diffraction
  - 'pg' : polarization gating

  Not all methods support all nonlinear processes. In that case a ValueError will be raised.

- **parameters are described in the documentation of the specific** (*Additional*) –
- **methods.** (*PNPS*) –

**class** pypret.pnps.**FROG**(*pulse*, *process*)

Implements frequency-resolved optical gating [Kane1993] [Trebino2000].

**__init__**(*pulse*, *process*)

Creates the instance.

**Parameters**

- **pulse** (*Pulse instance*) – The pulse object that defines the simulation grid.
- **process** (*str*) – The nonlinear process used in the PNPS method.

**method = 'frog'**

**parameter_name = 'delay'**

**parameter_unit = 's'**

**class** pypret.pnps.**IFROG**(*pulse*, *process*)

Implements the interferometric frequency-resolved optical gating method[1].

**__init__**(*pulse*, *process*)

Creates the instance.

**Parameters**

- **pulse** (*Pulse instance*) – The pulse object that defines the simulation grid.
- **process** (*str*) – The nonlinear process used in the PNPS method.

**mask**(*tau*)

**method = 'ifrog'**

**parameter_name = 'tau'**

**parameter_unit = 's'**

[1] G. Stibenz and G. Steinmeyer, "Interferometric frequency-resolved optical gating," Opt. Express 13, 2617-2626 (OSA, 2005).

**class** pypret.pnps.**TDP**(*pulse*, *process*, *center*, *width*)

Implements a variant of time-domain ptychography. This version is self-referenced and works like FROG except that in one arm of the correlator the bandwidth of the pulse is heavily filtered [Witting2016]. Other variants are not directly supported by this class.

**__init__**(*pulse*, *process*, *center*, *width*)

Creates the instance.

**Parameters**

- **pulse** (*Pulse instance*) – The pulse object that defines the simulation grid.
- **process** (*str*) – The nonlinear process used in the PNPS method.
- **center** (*float*) – The center wavelength of the bandwidth filter in m.
- **width** (*float*) – The width (FWHM) of the bandwidth filter in m.

**method = 'tdp'**

**parameter_name = 'delay'**

**parameter_unit = 's'**

**class** pypret.pnps.**DSCAN**(*pulse*, *process*, *material*)

Implements the dispersion scan method [Miranda2012a] [Miranda2012b].

Not implemented in the public version of the code. Please contact us if you want to use pypret for d-scan measurements.

**__init__**(*pulse*, *process*, *material*)

Initialize self. See help(type(self)) for accurate signature.

**method = 'dscan'**

**parameter_name = 'insertion'**

**parameter_unit = 'm'**

**class** pypret.pnps.**MIIPS**(*pulse*, *process*, *alpha*, *gamma*)

Implements the multiphoton intrapulse interference phase scan method (MIIPS) [Lozovoy2004] [Xu2006].

**__init__**(*pulse*, *process*, *alpha*, *gamma*)

Creates the instance.

**Parameters**

- **pulse** (*Pulse instance*) – The pulse object that defines the simulation grid.
- **process** (*str*) – The nonlinear process used in the PNPS method.
- **alpha** (*float*) – The amplitude of the phase pattern (in rad).
- **gamma** (*float*) – The frequency of the phase pattern in Hz.

**mask**(*delta*)

**method = 'miips'**

**parameter_name = 'delta'**

**parameter_unit = 'rad'**

### 3.3.2 API

**class** pypret.pnps.**BasePNPS**(*pulse*, *process*, ***kwargs*)

The PNPS base class

**__init__**(*pulse*, *process*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

**calculate**(*spectrum*, *parameter*)

Calculates the PNPS signal S_mk and the PNPS trace T_mn.

**Parameters**

- **spectrum** (*1d-array*) – The pulse spectrum for which the PNPS trace is calculated.
- **parameter** (*scalar or 1d-array*) – The PNPS parameter (array) for which the PNPS trace is calculated.

**Returns** Returns the calculated PNPS trace over the frequency `self.process_w`. If parameter was a scalar a 1d-array is returned. If it was a 1d-array a 2d-array is returned where the parameter runs along the first axis and the frequency along the second.

**Return type** 1d- or 2d-array

**gradient**(*Smk2*, *parameter*)

Calculates the gradient _n Z_m.

**intermediate**(*parameter*)

Returns intermediate results as stored by the instance.

**measure**(*Sk*)

Simulates the measurement process.

Note that we deal with the spectrum over the frequency! For retrieving from actual data we need to rescale this by lambda^2.

**method = None**

**parameter_name = ''**

**parameter_unit = ''**

**process = None**

**scheme**

**trace**

Returns the last calculated trace as a MeshData object.

**class** pypret.pnps.**CollinearPNPS**(*pulse*, *process*, ***kwargs*)

Implements collinear methods: d-scan, iFROG, etc.

**class** pypret.pnps.**NoncollinearPNPS**(*pulse*, *process*, ***kwargs*)

Implements non-collinear methods: FROG, TDP, etc.

## 3.4 pypret.retrieval package

This module provides classes to calculate parametrized nonlinear process spectra (PNPS), such as frequency-resolved optical gating (FROG), interferometric FROG (iFROG), dispersion scan (d-scan), time-domain ptychography (TDP) and pulse-shaper assisted methods such as multiphoton intrapulse interference phase scan (MIIPS).

The code follows the notation used in [Geib2019] and its supplement.

Currently only the abovementioned methods are implemented. But the code is written in such way that including new pulse measurement methods is very easy. If it is a method using a collinear nonlinearity, subclass from *CollinearPNPS*, otherwise from *NoncollinearPNPS*.

In the collinear case only *self.mask(parameter)* has to be implemented which calculates the used linear parametrization operator. In the non-collinear case the function *_calculate* has to be implemented which calculates and returns the PNPS trace `T_mn` and the PNPS signal `S_mk`.

### 3.4.1 Retrieval algorithms

`pypret.retrieval.retriever.`**`Retriever`**(*pnps: pypret.pnps.BasePNPS*, *method: str = 'copra'*, *maxiter=300*, *maxfev=None*, *logging=False*, *verbose=False*, ***kwargs*) → pypret.retrieval.retriever.BaseRetriever

Creates a retriever instance.

**Parameters**

- **pnps** (*PNPS*) – A PNPS instance that is used to simulate a PNPS measurement.
- **method** (*str, optional*) –

  **Type of solver. Should be one of**

  - 'copra' `(see here)`
  - 'gpa' `(see here)`
  - 'gp-dscan' `(see here)`
  - 'pcgpa' `(see here)`
  - 'pie' `(see here)`
  - 'lm' `(see here)`
  - 'bfgs' `(see here)`
  - 'de' `(see here)`
  - 'nelder-mead' `(see here)`

  'copra' is the default choice.
- **maxiter** (*int, optional*) – The maximum number of algorithm iterations. The default is 300.
- **maxfev** (*int, optional*) – The maximum number of function evaluations. If given, the algorithms stop before this number is reached. Not all algorithms support this feature. Default is `None`, in which case it is ignored.
- **logging** (*bool, optional*) – Stores trace errors and pulses over the iterations if supported by the retriever class. Default is *False*.
- **verbose** (*bool, optional*) – Prints out trace errors during the iteration if supported by the retriever class. Default is *False*.

**class** `pypret.retrieval.step_retriever.`**`COPRARetriever`**(*pnps*, *alpha=0.25*, ***kwargs*)

This module implements the common pulse retrieval algorithm [Geib2019].

**`__init__`**(*pnps*, *alpha=0.25*, ***kwargs*)

For a full documentation of the arguments see `Retriever`.

**Parameters** **alpha** (*float, optional*) – Scales the step size in the global stage of COPRA. Higher values mean potentially faster convergence but less accuracy. Lower values provide higher accuracy for the cost of speed. Default is 0.25.

**method = 'copra'**

**class** pypret.retrieval.step_retriever.**PCGPARetriever**(*pnps*, *decomposition='power'*, ***kwargs*)

This class implements the principal components generalized projections algorithm (PCGPA) for SHG-FROG.

We follow the algorithm as described in [Kane1999] but use the PNPS formalism from [Geib2019] and some minor modifications:

- it supports both the singular value decomposition and the power method to find/approximate the largest eigenvector.
- the projection includes the scaling factor μ. This makes the method robust against initial guesses with the wrong magnitude. It should have no adverse effect.

**__init__**(*pnps*, *decomposition='power'*, ***kwargs*)

For a full documentation of the arguments see `Retriever`.

**Parameters** **decomposition** (*str, optional*) – It specifies how the FROG signal is decomposed. If *power* (the default) the power method is used to find the largest eigenvalue. If *svd* a full singular value decomposition is performed. This is potentially much slower but more accurate.

**method = 'pcgpa'**

**supported_schemes = ['shg-frog']**

**class** pypret.retrieval.step_retriever.**GPARetriever**(*pnps*, *step_size='exact'*, ***kwargs*)

Implements the classical generalized projections algorithm for SHG-FROG as described in [DeLong1994] and [Trebino2000].

As far as I know the determination of the step size in GPA is not made explicit in the publications. It is usually done in a line search. In this implementation we offer three different options:

- an exact line search using a Brent style minimizer
- a backtracking (inexact) line search using the Armijo-Goldstein condition with c=0.5 and tau=0.5.
- the same heuristic choice for the step size used in copra.

The last method is the fastest, but as the first is the classic choice for GPA, it is the default.

**__init__**(*pnps*, *step_size='exact'*, ***kwargs*)

For a full documentation of the arguments see `Retriever`.

**Parameters** **step_size** (*str, optional*) – Specifies how the step size of the gradient step in GPA is determined. Default is *exact* which performs an exact line search. *inexact* performs a backtracking line search and *copra* uses the ad-hoc estimates for the step size used in COPRA.

**method = 'gpa'**

**supported_schemes = ['shg-frog']**

**class** pypret.retrieval.step_retriever.**PIERetriever**(*pnps*, *logging=False*, *verbose=False*, ***kwargs*)

This class implements a pulse retrieval algorithm for SHG-FROG based on the ptychographical iterative engine (PIE). It is based on the paper [Sidorenko2016] and its erratum [Sidorenko2017].

We modified the algorithm to include the scaling factor μ in the projection. This makes the method robust against initial guesses with the wrong magnitude. It should have no adverse effect.

**method = 'pie'**

**supported_schemes = ['shg-frog']**

**class** pypret.retrieval.step_retriever.**GPDSCANRetriever**(*pnps*, *logging=False*, *verbose=False*, ***kwargs*)

This class implements a pulse retrieval algorithm for SHG and THG d-scan based on the paper [Miranda2017].

In our tests we found that it does not converge in the noiseless case. In other words the global solution to the least-squares problem is not a fixed point of the iteration.

**method = 'gp-dscan'**

**supported_schemes = ['shg-dscan', 'thg-dscan']**

**class** pypret.retrieval.nlo_retriever.**LMRetriever**(*pnps*, *ftol=1e-08*, *xtol=1e-08*, *gtol=1e-08*, *lm_verbose=0*, ***kwargs*)

Implements pulse retrieval based on the Levenberg-Marquadt algorithm.

This is an efficient nonlinear least-squares solver, however, it will still be *very* slow for large pulses (N > 256). The reason is that the (MN x N) Jacobian is evaluated using numerical differentiation.

The recommendation is to use this method either on small problems or to refine or verify solutions provided by a different algorithm.

**__init__**(*pnps*, *ftol=1e-08*, *xtol=1e-08*, *gtol=1e-08*, *lm_verbose=0*, ***kwargs*)

For a full documentation of the arguments see `Retriever`.

For the documentation of *ftol*, *xtol*, *gtol* see the documentation of `scipy.optimize.least_squares()`. They are passed directly to the optimizer. If you want to run the optimizer for a fixed number of iterations, set all values to 1e-14 to effectively disable the stopping criteria.

**method = 'lm'**

**class** pypret.retrieval.nlo_retriever.**NMRetriever**(*pnps*, *logging=False*, *verbose=False*, ***kwargs*)

This retriever uses the gradient-free Nelder-Mead algorithm.

**method = 'nm'**

**class** pypret.retrieval.nlo_retriever.**DERetriever**(*pnps*, *logging=False*, *verbose=False*, ***kwargs*)

This retriever uses the gradient-free differential evolution algorithm.

It tries to match the parameters described in [Escoto2018] as far as they are mentioned. No further effort was made to optimize them. If you are interested in using DE as a pulse retrieval algorithm you are advised to study the documentation at `scipy.optimize.differential_evolution()`.

The initial population in our implementation is based on the provided guess with added complex, Gaussian noise of 5% of the maximum amplitude. In our tests we saw no convergence when starting from completely random initial guesses.

**method = 'de'**

**class** pypret.retrieval.nlo_retriever.**BFGSRetriever**(*pnps*, *logging=False*, *verbose=False*, ***kwargs*)

This retriever uses the BFGS algorithm with numerical differentiation.

**method = 'bfgs'**

### 3.4.2 API

**class** `pypret.retrieval.retriever.`**`BaseRetriever`**(*pnps*, *logging=False*, *verbose=False*, ***kwargs*)

The abstract base class for pulse retrieval.

This class implements common functionality for different retrieval algorithms.

**`__init__`**(*pnps*, *logging=False*, *verbose=False*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

**`method = None`**

**`result`**(*pulse_original=None*, *full=True*)
Analyzes the retrieval results in one retrieval instance and processes it for plotting or storage.

**`retrieve`**(*measurement*, *initial_guess*, *weights=None*, ***kwargs*)
Retrieve pulse from `measurement` starting at `initial_guess`.

**Parameters**

- **`measurement`** (`MeshData`) – A MeshData instance that contains the PNPS measurement. The first axis has to correspond to the PNPS parameter, the second to the frequency. The data has to be the measured _intensity_ over the frequency (not wavelength!). The second axis has to match exactly the frequency axis of the underlying PNPS instance. No interpolation is done.
- **`initial_guess`** (*`1d-array`*) – The spectrum of the pulse that is used as initial guess in the iterative retrieval.
- **`weights`** (*`1d-array`*) – Weights that are attributed to the measurement for retrieval. In the case of (assumed) Gaussian uncertainties with standard deviation sigma they should correspond to 1/sigma. Not all algorithms support using the weights.
- **`kwargs`** (*`dict`*) – Can override retrieval options specified in `__init__()`.

#### Notes

This function provides no interpolation or data processing. You have to write a retriever wrapper for that purpose.

**`supported_schemes = None`**

**`trace_error`**(*spectrum*, *store=True*)
Calculates the trace error from the pulse spectrum.

**class** `pypret.retrieval.step_retriever.`**`StepRetriever`**(*pnps*, *logging=False*, *verbose=False*, ***kwargs*)

## 3.5 pypret.pulse_error

This module implements testing procedures for retrieval algorithms.

`pypret.pulse_error.`**`best_constant_phase`**(*E*, *E0*)
Finds `c` with `|c| = 1` so that `sum(abs2(c * y1 - y2))` is minimal.

Uses an analytic solution.

`pypret.pulse_error.`**`optimal_rms_error`**`(`*w*, *E*, *E0*`)`

Calculates the RMS error of two arrays, ignoring scaling, constant and linear phase of one of them.

Formally it calculates the minimal error:

```
R = sqrt(|rho * exp(i*(x*a + b)) * y1 - y2|^2 / |y2|^2)
```

with respect to rho, a and b. If additionally `conjugation = True` then the error for conjugate(y1) is calculated and the best transformation of y1 is also returned.

`pypret.pulse_error.`**`pulse_error`**`(`*E*, *E0*, *ft*, *dot_ambiguity=False*, *spectral_shift_ambiguity=False*`)`

Calculates the normalized rms error between two pulse spectra while taking into account the retrieval ambiguities.

One step in *optimal_rms_error* (the determination of the initial bracket) could probably be more efficient, see [Dorrer2002]). We use the less elegant but maybe more straightforward way of simply sampling the range for a bracket that encloses a minimum.

**Parameters**

- **`E0`** (*`E,`*) – Complex-valued arrays that contain the spectra of the pulses. `E` will be matched against `E0`.
- **`ft`** (*`FourierTransform instance`*) – Performs Fourier transforms on the pulse grid.
- **`dot_ambiguity`** (*`bool, optional`*) – Takes the direction of time ambiguity into account. Default is `False`.
- **`spectral_shift_ambiguity`** (*`bool, optional`*) – Takes the spectral shift ambiguity into account. Default is `False`.

## 3.6 pypret.io package

A subpackage that provides Python object persistence in HDF5 files.

It was written to make the storage of arbitrary nested Python structures in the exchangable HDF5 format easy. Its main purpose is to easily add persistence to existing numerical or data analysis codes.

While the files itself are plain HDF5 and can be read in any language supporting HDF5, the format is not compatible to Matlab's own file format. If you are searching for such a solution look at the hdf5storage package.

### 3.6.1 Usage

The module exports a `save()` function that stores arbitrary structures of Python and NumPy data types. For example

```
>>> x = {'data': [1, 2, 3], 'xrange': np.arange(5, dtype=np.uint8)}
>>> io.save(x, "test.hdf5")
```

This function should suffice for most needs as long as only standard types are used. The `load()` function loads these files and restores the structure and the types of the data:

```
>>> io.load("test.hdf5")
{'data': [1, 2, 3], 'xrange': array([0, 1, 2, 3, 4], dtype=uint8)}
```

### 3.6.2 Custom Objects

If you are using objects as simple containers without functionality you may consider using the SimpleNamespace class from the `types` module of the standard library. The advantage is that io knows how to handle it.:

```
from types import SimpleNamespace
a = SimpleNamespace(name="my object", data=np.arange(5))
a.data2 = np.arange(10)
copra.save(a)
```

If your objects are containers with methods but without a custom `__init__()` the simplest way is to inherit or mix-in the `IO` class:

```
class Data(io.IO):
    x = 1

    def squared(self):
        return self.x * self.x
```

When using the `IO` class by default all instance attributes are stored and loaded. More flexibility can be achieved by specifying `_io`-attributes of your custom class.

**_io_store** [list of str or None, optional] Specify the the instance attributes that are stored exclusively. Acts as a whitelist. If `None` all instance attributes are stored. Default is `None`.

**_io_store_not** [list of str or None, optional] Specify which instance attributes are not stored. Acts as a blacklist. If `None` no blacklisting is done.

If you want to add attributes to storage you can call the `_io_add_to_storage(key)` method on your instance. The IO class initalizes the instance without calling `__init__()`. Instead `__new__()` is called on the class and afterwards the `_post_init()` method which subclasses can implement. A fully working example of a class is the following (reduced from copra.FourierTransform):

```
class Grid(io.IO):
    _io_store = ['N', 'dx', 'x0']

    def __init__(self, N, dx, x0=0.0):
        # This is _not_ called upon loading from storage
        self.N = N
        self.dx = dx
        self.x0 = x0
        self._post_init()

    def _post_init(self):
        # this is called upon loading from storage
        # calculate the grids
        n = np.arange(self.N)
        self.x = self.x0 + n * self.dx
```

In this example the object can be exactly reproduced upon loading but only a minimal amount of storage is required.

If you want to implement your own storage interface for a custom object you should inherit from `IO` and implement your own `to_dict()` and `from_dict()` methods. Look at the implementation of the default in `IO` to understand their behavior.

### 3.6.3 File Format

The file format this module uses is a straightforward mapping of Python types to the HDF5 data structure. Dictionaries and objects are mapped to HDF5 groups, numpy arrays use h5py's type translation. Iterables are converted to groups by introducing artificial keys of the type `idx_%d`. This is rather inefficient which explains why the module should not be used to store large numerical arrays as a Python list. To store the type information it uses an HDF5 attribute `__class__`. Furthermore, for scalars the attribute `__dtype__` and for strings the attribute `__encoding__` are additionally used.

In conclusion, nested structures of Python types stored with this package are not suitable for exchanging. Dictionaries of numerical data stored with this package can be easily opened with any program that supports HDF5.

### 3.6.4 Public interface

**class** `pypret.io.options.`**`HDF5Options`**

A class that handles the correct HDF5 options for different data sets.

The reason is simply that native HDF5 compression will actually increase the file size for small arrays (< 300 bytes). This class selects different HDF5 options based on the dataset over the method `__call__`. It can be subclassed to support more sophisticated selection strategies.

**`__init__`**`()`

Initialize self. See help(type(self)) for accurate signature.

**`copy`**`()`

`pypret.io.`**`save`**`(`*val*, *path*, *archive=False*, *options=<pypret.io.options.HDF5Options object>*`)`

Saves an object in an HDF5 file.

**Parameters**

- **val** (*object*) – Any Python value that is made up of storeable instances. Those are built-in types, numpy datatypes and types with custom handlers.
- **path** (*str or Path instance*) – Save path of the HDF5 file. Existing files will be overwritten!
- **archive** (*bool, optional*) – If `True` will compress the whole hdf5 file. This is useful when dealing with (many) small HDF5 files as those contain significant overhead.
- **options** (*HDF5Options instance, optional*) – The HDF5 options that will be used for saving. Defaults to the global options instance *DEFAULT_OPTIONS*.

`pypret.io.`**`load`**`(`*path*, *obj=None*, *archive=None*`)`

Reads a possibly compressed HDF5 file.

If archive is `None` it is retrieved with python-magic.

**class** `pypret.io.`**`IO`**

Provides an interface for saving to and loading from a HDF5 file.

This class can be mixed-in to easily add persistence to your existing Python classes. By default all attributes of an object will be stored. Upon loading these attributes will be loaded and *__init__* will *not* be called.

Often a better way is to store only the necessary attributes by giving a list of attribute names in the private attribute *_io_store*. Then you have to overwrite the *_post_init()* method that initializes your object from these stored attributes. It is usually also be called at the end of the original *__init__* and should not mean extra effort.

Lastly, you can simply overwrite *load_from_dict* to implement a completely custom loader.

**`classmethod from_dict`**`(`*attrs*`)`

**classmethod load**(*path*)

**classmethod load_from_group**(*group*)

**save**(*path*, *archive=False*, *options=<pypret.io.options.HDF5Options object>*)

**save_to_group**(*g*, *name*)

**to_dict**()

**update**(*path*)

**update_from_dict**(*attrs*)

**update_from_group**(*group*)

## 3.6.5 Custom handlers

Implements functions that handle the serialization of types and classes.

Type handlers store and load objects of exactly that type. Instance handlers work also work for subclasses of that type.

The instance handlers are processed in the order they are stored. This means that if an object is an instance of several handled classes it will not raise an error and will be handled by the first matching handler in the OrderedDict.

**class** pypret.io.handlers.**Handler**

**classmethod create_dataset**(*data*, *level*, *name*, ***kwargs*)

**classmethod create_group**(*level*, *name*, *options*)

**classmethod get_type**(*level*)

**classmethod is_dataset**()

**classmethod is_group**()

**level_type = 'dataset'**

**classmethod load_from_level**(*level*, *obj=None*)
The loader that has to be implemented by subclasses.

**classmethod save_to_level**(*val*, *level*, *options*, *name*)
A generic wrapper around the custom save method that each handler implements. It creates a dataset or a group depending on the *level_type* class attribute and sets the *__class__* attribute correctly. For more flexibility subclasses can overwrite this method.

**class** pypret.io.handlers.**TypeHandler**
Handles data of a specific type or class.

**casting = {'builtins.NoneType': <class 'NoneType'>, 'builtins.bool': <class 'bool'>,**

**classmethod register**(*t*)

**types = []**

**class** pypret.io.handlers.**InstanceHandler**
Handles all instances of a specific (parent) class.

If an instance is subclass to several classes for which a handler exists, no error will be raised (in contrast to TypeHandler). Rather, the first match in the global instance_saver_handlers OrderedDict will be used.

**casting = {'pypret.fourier.FourierTransform': <class 'pypret.fourier.FourierTransform**

**instances = []**

**classmethod register**(*t*)

## 3.7 pypret.lib module

Miscellaneous helper functions

These functions fulfill small numerical tasks used in several places in the package.

`pypret.lib.`**`abs2`**(*x*)
Calculates the squared magnitude of a complex array.

`pypret.lib.`**`arglimit`**(*y*, *threshold=0.001*, *padding=0.0*, *normalize=True*)
Returns the first and last index where *y >= threshold * max(abs(y))*.

`pypret.lib.`**`as_list`**(*x*)
Try to convert argument to list and return it.

Useful to implement function arguments that could be scalar values or lists.

`pypret.lib.`**`best_scale`**(*E*, *E0*)
Scales rho so that:

```
sum (rho * |E| - |E0|)^2
```

is minimal.

`pypret.lib.`**`build_coords`**(**axes*)
Builds a coordinate array from the axes.

`pypret.lib.`**`edges`**(*x*)
Calculates the edges of the array elements.

Assuming that the input array contains the midpoints of a supposed data set, the function returns the (N+1) edges of the data set points.

`pypret.lib.`**`find`**(*x*, *condition*, *n=1*)
Return the index of the nth element that fulfills the condition.

`pypret.lib.`**`fwhm`**(*x*, *y*)
Calculates the full width at half maximum of the distribution described by (x, y).

`pypret.lib.`**`gaussian`**(*x*, *x0=0.0*, *sigma=1.0*)
Calculates a Gaussian function with center `x0` and standard deviation `sigma`.

`pypret.lib.`**`jit`**(*pyfunc=None*, ***kwargs*)

`pypret.lib.`**`limit`**(*x*, *y=None*, *threshold=0.001*, *padding=0.25*, *extend=True*)
Returns the maximum x-range where the y-values are sufficiently large.

**Parameters**

- **x** (*array_like*) – The x values of the graph.
- **y** (*array_like, optional*) – The y values of the graph. If *None* the maximum range of *x* is used. That is only useful if *padding > 0*.
- **threshold** (*float*) – The threshold relative to the maximum of *y* of values that should be included in the bracket.
- **padding** (*float*) – The relative padding on each side in fractions of the bracket size.
- **extend** (*bool, optional*) – Signals if the returned range can be larger than the values in `x`. Default is *True*.

**Returns** **xl, xr** – Lowest and biggest value of the range.

**Return type** float

`pypret.lib.`**`marginals`**(*data*, *normalize=False*, *axes=None*)

Calculates the marginals of the data array.

axes specifies the axes of the marginals, e.g., the axes on which the sum is projected.

If axis is None a list of all marginals is returned.

`pypret.lib.`**`mask_phase`**(*x*, *amp*, *phase*, *threshold=0.001*)

`pypret.lib.`**`mean`**(*x*, *y*)

Calculates the mean of the distribution described by (x, y).

`pypret.lib.`**`norm`**(*x*)

Calculates the L2 or Euclidian norm of array `x`.

`pypret.lib.`**`norm2`**(*x*)

Calculates the squared L2 or Euclidian norm of array `x`.

`pypret.lib.`**`nrms`**(*x*, *y*)

Calculates the normalized rms error between `x` and `y`.

The convention for normalization varies. Here we use:

```
max |y|
```

as normalization.

`pypret.lib.`**`phase`**(*x*)

The phase of a complex array.

`pypret.lib.`**`rescale`**(*x*, *window=[0.0, 1.0]*)

Rescales a numpy array to the range specified by `window`.

Default is [0, 1].

`pypret.lib.`**`retrieval_report`**(*res*)

Simple helper that prints out important information from the retrieval result object.

`pypret.lib.`**`rms`**(*x*, *y*)

Calculates the root mean square (rms) error between `x` and `y`.

`pypret.lib.`**`standard_deviation`**(*x*, *y*)

Calculates the standard deviation of the distribution described by (x, y).

`pypret.lib.`**`variance`**(*x*, *y*)

Calculates the variance of the distribution described by (x, y).

## 3.8 pypret.frequencies module

This module handles conversion between frequency units.

The supported units and their shorthands are:

- wl : wavelength in meter
- om: angular frequency in rad/s
- f: frequency in 1/s
- k: angular wavenumber in rad/m

The conversion functions have the form *shorthand2shorthand* which is not pythonic but very short. A more pythonic conversion can be achieved by using the *convert* function

```
>>> convert(x, 'wl', 'om')
```

The shorthands will be used throughout the package to identify frequency units.

The functions in this module should be used wherever a frequency convention is necessary to avoid mistakes and make the code more expressive.

`pypret.frequencies.`**`convert`**`(`*x*, *unit1*, *unit2*`)`

Convert between two frequency units.

**Parameters**

- **x** (*float or array_like*) – Numerical value or array that should be converted.
- **unit2** (*unit1,*) – Shorthands for the original unit (*unit1*) and the destination unit (*unit2*).

**Returns** The converted numerical value or array. It will always be a copy, even if *unit1* == *unit2*.

**Return type** float or array_like

### Notes

Unit shorthands can be any of *wl* : wavelength in meter *om* : angular frequency in rad/s *f* : frequency in 1/s *k* : angular wavenumber in rad/m

`pypret.frequencies.`**`f2k`**`(`*f*`)`

`pypret.frequencies.`**`f2om`**`(`*f*`)`

`pypret.frequencies.`**`f2wl`**`(`*f*`)`

`pypret.frequencies.`**`k2f`**`(`*k*`)`

`pypret.frequencies.`**`k2om`**`(`*k*`)`

`pypret.frequencies.`**`k2wl`**`(`*k*`)`

`pypret.frequencies.`**`om2f`**`(`*om*`)`

`pypret.frequencies.`**`om2k`**`(`*om*`)`

`pypret.frequencies.`**`om2wl`**`(`*om*`)`

`pypret.frequencies.`**`wl2f`**`(`*wl*`)`

`pypret.frequencies.`**`wl2k`**`(`*wl*`)`

`pypret.frequencies.`**`wl2om`**`(`*wl*`)`

## 3.9 pypret.material module

This module provides classes to calculate the refractive index based on Sellmeier equations.

This is required to correctly model d-scan measurements.

Currently only very few materials are implemented. But more should be easy to add. If the refractive index is described by formula 1 or 2 from refractiveindex.info you can simply instantiate *SellmeierF1* or *SellmeierF2*. If not, inherit from BaseMaterial and implement the *self._func* method.

### 3.9.1 Available materials

`pypret.material.`**`BK7 = <pypret.material.SellmeierF2 object>`**

Material instance describing N-BK7 (SCHOTT).

The data was taken from refractiveindex.info

`pypret.material.`**`FS = <pypret.material.SellmeierF1 object>`**

Material instance describing fused silica (fused quartz).

The data was taken from refractiveindex.info

### 3.9.2 Base classes

**`class`** `pypret.material.`**`BaseMaterial`**(*coefficients*, *freq_range*, *scaling=1000000.0*, *check_bounds=True*, *name=”*, *long_name=”*)

Abstract base class for dispersive materials.

**`__init__`**(*coefficients*, *freq_range*, *scaling=1000000.0*, *check_bounds=True*, *name=”*, *long_name=”*)

Creates a dispersive material.

**Parameters**

- **`coefficients`** (*`ndarray`*) – The Sellmeier coefficients.
- **`freq_range`** (*`iterable`*) – The wavelength range in which the Sellmeier equation is valid (given in m).
- **`check_bounds`** (*`bool, optional`*) – Specifies if the frequency argument should be checked on every evaluation to match the allowed range.
- **`scaling`** (*`float, optional`*) – Specifies the scaling of the Sellmeier formula. E.g., most Sellmeier formulas are defined in terms of µm (micrometer), whereas our function interface works in meter. In that case the scaling would be *1e6*. Default is *1.0e6*.

**`k`**(*x*, *unit=’wl’*)

The wavenumber in the material in rad / m.

**`n`**(*x*, *unit=’wl’*)

The refractive index at frequency *x* specified in units *unit*.

**`class`** `pypret.material.`**`SellmeierF1`**(*coefficients*, *freq_range*, *scaling=1000000.0*, *check_bounds=True*, *name=”*, *long_name=”*)

Defines a dispersive material via a specific Sellmeier equation.

This subclass supports materials with a Sellmeier equation of the form:

```
n^2(l) - 1 = c1 + c2 * l^2 / (l2 - c3^2) + ...
```

This is formula 1 from refractiveindex.info [DispersionFormulas].

**`class`** `pypret.material.`**`SellmeierF2`**(*coefficients*, *freq_range*, *scaling=1000000.0*, *check_bounds=True*, *name=”*, *long_name=”*)

Defines a dispersive material via a specific Sellmeier equation.

This subclass supports materials with a Sellmeier equation of the form:

```
n^2(l) - 1 = c1 + c2 * l^2 / (l2 - c3) + ...
```

This is formula 2 from refractiveindex.info [DispersionFormulas].

## 3.10 pypret.mesh_data module

This module implements an object for dealing with two-dimensional data.

**class** `pypret.mesh_data.`**MeshData**(*data*, **axes*, *uncertainty=None*, *labels=None*, *units=None*)

Bases: `pypret.io.io.IO`

**`__init__`**(*data*, **axes*, *uncertainty=None*, *labels=None*, *units=None*)

Creates a MeshData instance.

**Parameters**

- **data** (*ndarray*) – A at least two-dimensional array containing the data.
- ***axes** (*ndarray*) – Arrays specifying the coordinates of the data axes. Must be given in indexing order.
- **uncertainty** (*ndarray*) – An ndarray of the same size as *data* that contains some measure of the uncertainty of the meshdata. E.g., it could be the standard deviation of the data.
- **labels** (*list of str, optional*) – A list of strings labeling the axes. The last element labels the data itself, e.g. `labels` must have one more element than the number of axes.
- **units** (*list of str, optional*) – A list of unit strings.

**autolimit**(**axes*, *threshold=0.01*, *padding=0.25*)

Limits the data based on the marginals.

**copy**()

Creates a copy of the MeshData instance.

**flip**(**axes*)

Flips the data on the specified axes.

**interpolate**(*axis1=None*, *axis2=None*, *degree=2*, *sorted=False*)

Interpolates the data on a new two-dimensional, equidistantly spaced grid.

**limit**(**limits*, *axes=None*)

Limits the data range of this instance.

**Parameters**

- ***limits** (*tuples*) – The data limits in the axes as tuples. Has to match the dimension of the data or the number of axes specified in the *axes* parameter.
- **axes** (*tuple or None*) – The axes in which the limit is applied. Default is *None* in which case all axes are selected.

**marginals**(*normalize=False*, *axes=None*)

Calculates the marginals of the data.

axes specifies the axes of the marginals, e.g., the axes on which the sum is projected.

**ndim**

Returns the dimension of the data as integer.

**normalize**()

Normalizes the maximum of the data to 1.

**scale**(*scale*)

**shape**
Returns the shape of the data as a tuple.

## 3.11 pypret.graphics module

This module implements several helper routines for plotting.

**class** `pypret.graphics.`**MeshDataPlot**(*mesh_data*, *plot=True*, ***kwargs*)
Bases: `object`

**__init__**(*mesh_data*, *plot=True*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

**plot**(*show=True*)

**show**()

**class** `pypret.graphics.`**PulsePlot**(*pulse*, *plot=True*, ***kwargs*)
Bases: `object`

**__init__**(*pulse*, *plot=True*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

**plot**(*xaxis='wavelength'*, *yaxis='intensity'*, *limit=True*, *oversampling=False*, *phase_blanking=False*, *phase_blanking_threshold=0.001*, *show=True*)

`pypret.graphics.`**plot_complex**(*x*, *y*, *ax*, *ax2*, *yaxis='intensity'*, *limit=False*, *phase_blanking=False*, *phase_blanking_threshold=0.001*, *amplitude_line='r-'*, *phase_line='b-'*)

`pypret.graphics.`**plot_meshdata**(*ax*, *md*, *cmap='nipy_spectral'*)

# Bibliography

[Geib2019] Nils C. Geib, Matthias Zilk, Thomas Pertsch, and Falk Eilenberger, “Common pulse retrieval algorithm: a fast and universal method to retrieve ultrashort pulses,” Optica 6, 495-505 (2019)

[Lozovoy2004] V. V. Lozovoy, I. Pastirk and M. Dantus, “Multiphoton intrapulse interference. IV. Ultrashort laser pulse spectral phase characterization and compensation,” Opt. Lett. 29, 775-777 (OSA, 2004).

[Xu2006] B. Xu, J. M. Gunn, J. M. D. Cruz, V. V. Lozovoy and M. Dantus, “Quantitative investigation of the multiphoton intrapulse interference phase scan method for simultaneous phase measurement and compensation of femtosecond laser pulses,” J. Opt. Soc. Am. B 23, 750-759 (OSA, 2006).

[Miranda2012a] M. Miranda, T. Fordell, C. Arnold, A. L’Huillier and H. Crespo, “Simultaneous compression and characterization of ultrashort laser pulses using chirped mirrors and glass wedges,” Opt. Express 20, 688-697 (OSA, 2012).

[Miranda2012b] M. Miranda, C. L. Arnold, T. Fordell, F. Silva, B. Alonso, R. Weigand, A. L’Huillier and H. Crespo, “Characterization of broadband few-cycle laser pulses with the d-scan technique,” Opt. Express 20, 18732-18743 (OSA, 2012).

[Kane1993] D. J. Kane and R. Trebino, “Characterization of arbitrary femtosecond pulses using frequency-resolved optical gating,” IEEE J. Quant. Electron. 29, 571-579 (IEEE, 1993).

[Kane1999] D. J. Kane, “Recent progress toward real-time measurement of ultrashort laser pulses,” IEEE J. Quant. Electron. 35, 421-431 (IEEE, 1999).

[Trebino2000] R. Trebino, “Frequency-Resolved Optical Gating: The Measurement of Ultrashort Laser Pulses,” , (Springer US, 2000).

[Witting2016] T. Witting, D. Greening, D. Walke, P. Matia-Hernando, T. Barillot, J. P. Marangos and J. W. G. Tisch, “Time-domain ptychography of over-octave-spanning laser pulses in the single-cycle regime,” Opt. Lett. 41, 4218-4221 (OSA, 2016).

[Dorrer2002] C. Dorrer and I. A. Walmsley, “Accuracy criterion for ultrashort pulse characterization techniques: application to spectral phase interferometry for direct electric field reconstruction,” J. Opt. Soc. Am. B 19, 1019-1029 (OSA, 2002).

[Briggs1995] W. L. Briggs and v. E. Henson, “The DFT: an owners’ manual for the discrete Fourier transform,” (SIAM, 1995).

[Hansen2014] E. W. Hansen, “Fourier transforms: principles and applications,” (John Wiley & Sons, 2014).

[Trefethen2014] L. N. Trefethen and J. A. C. Weideman, “The exponentially convergent trapezoidal rule,” SIAM Review 56, 385-458 (2014).

[DispersionFormulas] http://refractiveindex.info/database/doc/Dispersion%20formulas.pdf

[Sidorenko2016] P. Sidorenko, O. Lahav, Z. Avnat and O. Cohen, “Ptychographic reconstruction algorithm for frequency-resolved optical gating: super-resolution and supreme robustness,” Optica 3, 1320-1330 (OSA, 2016).

[Sidorenko2017] P. Sidorenko, O. Lahav, Z. Avnat and O. Cohen, “Ptychographic reconstruction algorithm for frequency resolved optical gating: super-resolution and extreme robustness: erratum,” Optica 4, 1388-1389 (OSA, 2017).

[Miranda2017] M. Miranda, J. Penedones, C. Guo, A. Harth, M. Louisy, L. Neoričić, A. L’Huillier and C. L. Arnold, “Fast iterative retrieval algorithm for ultrashort pulse characterization using dispersion scans,” J. Opt. Soc. Am. B 34, 190-197 (OSA, 2017).

[DeLong1994] K. W. DeLong, B. Kohler, K. Wilson, D. N. Fittinghoff and R. Trebino, “Pulse retrieval in frequency-resolved optical gating based on the method of generalized projections,” Opt. Lett. 19, 2152-2154 (Optical Society of America, 1994)

[Escoto2018] E. Escoto, A. Tajalli, T. Nagy and G. Steinmeyer, “Advanced phase retrieval for dispersion scan: a comparative study,” J. Opt. Soc. Am. B 35, 8-19 (OSA, 2018).

[Diels2006] J.-C. Diels and W. Rudolph, “Ultrashort laser pulse phenomena,” 2nd ed. (Academic press, 2006)

# Python Module Index

# Index