Pyofss Documentation

Release 0.8-20120908

David Bolt (daibo)

September 08, 2012

CONTENTS

Pyofss allows optical component modules to be grouped together into a system. An optical field is propagated through the system and the evolution of the field viewed at each module.

CHAPTER

ONE

CONTENTS

1.1 Installation

Warning: Some of the following commands may require superuser privileges. Use:

```
$ sudo command
```

or:

```
$ su root
```

then run the commands.
Always be cautious when running with enhanced privileges!

Pyofss is available on Pypi and may be retrieved using the pip program:

```
$ aptitude install pip
$ pip install pyofss
```

Then import pyofss within scripts or in an interactive session:

```
>>> from pyofss import *
```

Note: If the required dependencies are not satisfied when installing pyofss, then manually install using either:

```
$ aptitude install python-numpy python-scipy python-matplotlib
```

or:

```
$ pip install numpy scipy matplotlib
```

The recommended versions are listed in the "requirements.txt" file within the pyofss package. Using this file, it is possible to automatically install all dependencies:

```
$ pip install -r requirements.txt
```

1.2 Pyofss tutorial

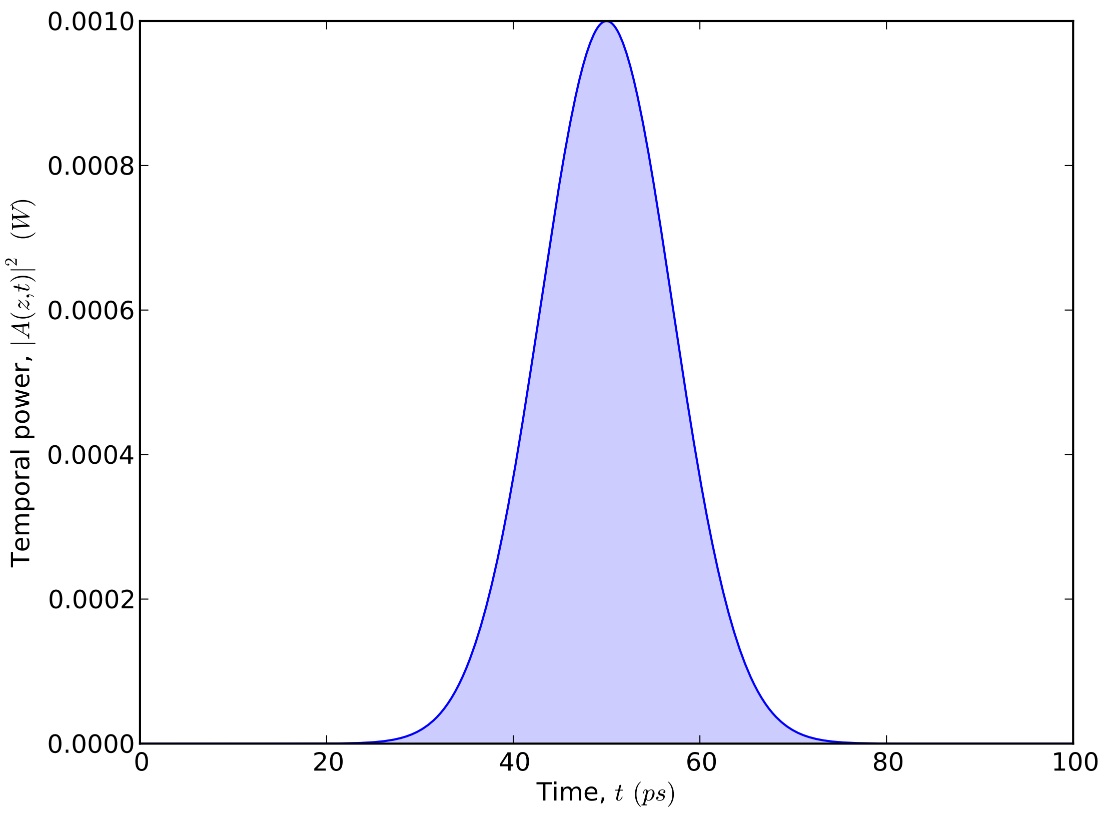
Release 0.8

Date September 08, 2012

Pyofss has been developed to be fully interactive. It allows the user to construct an optical fibre system, to get detailed information about the system, and to study the effects on an optical signal propagating through such a system.

Here is an example of a simple system consisting of a Gaussian pulse generator:

```
from pyofss import *
sys = System()
sys.add( Gaussian() )
sys.run()
single_plot( sys.domain.t, temporal_power(sys.field),
             labels['t'], labels['P_t'] )
```



1.2.1 System

Note: It is assumed that pyofss has been imported using

```
>>> from pyofss import *
```

Every pyofss simulation begins with a system. To generate a system with a default domain:

```
>>> sys = System()
```

which is equivalent to

```
>>> sys = System( Domain() )
```

Various modules may be added to the system such as pulse generators, fibres, and filters. Once a module has been generated, add it to the system. As an example, to construct a Gaussian pulse generator and add it to the system:

```
>>> gaussian = Gaussian()
>>> sys.add( gaussian )
```

It is also possible to directly add a module using:

```
>>> sys.add( Gaussian() )
```

Once all modules have been added, run the simulation

```
>>> sys.run()
```

This generates a field which is modified by each module it propagates through.

The output field may be accessed using:

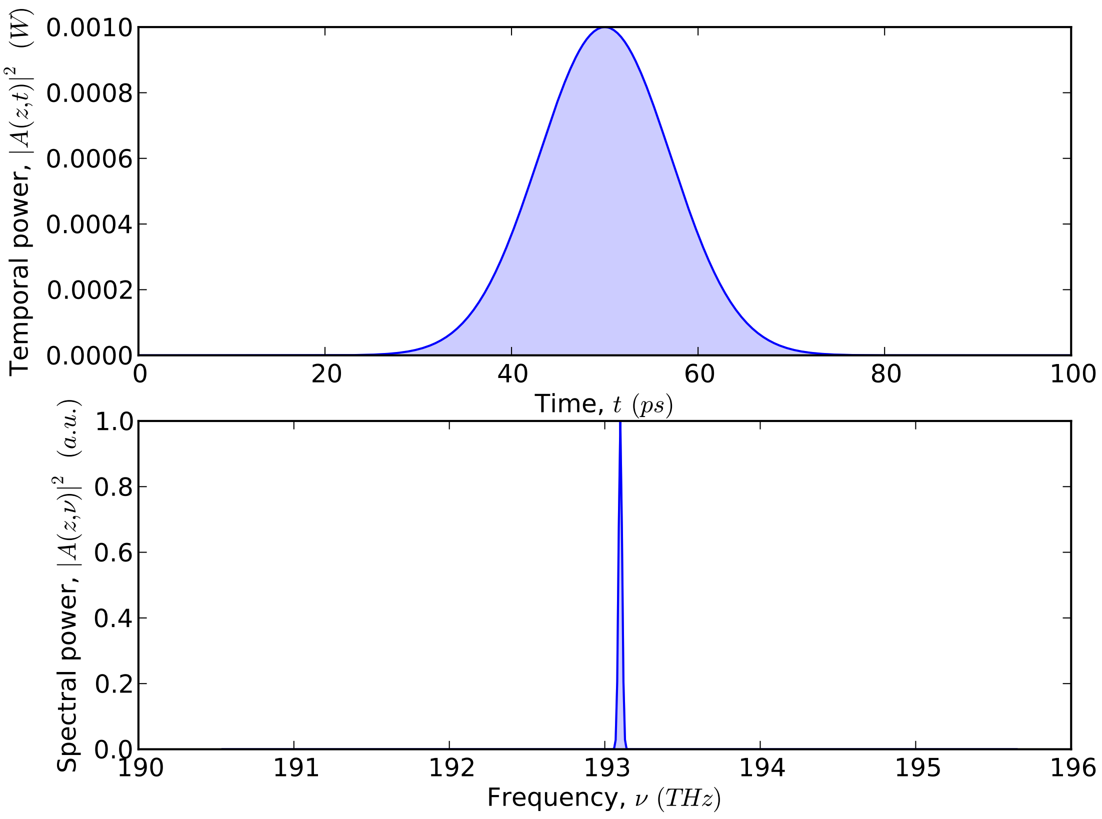
```
>>> sys.field
```

A field will generally be an array of complex values. There are utility functions which allow calculating the temporal and spectral power:

```
>>> P_t = temporal_power( sys.field )
>>> P_nu = spectral_power( sys.field )
```

There is also a plotter which allows a range of plots to be generated. As an example, the temporal and spectral power of the resulting field may be plotted in a "double" plot:

```
from pyofss import *
sys = System( Domain() )
sys.add( Gaussian() )
sys.run()
double_plot( sys.domain.t, temporal_power(sys.field),
             sys.domain.nu, spectral_power(sys.field, True),
             labels['t'], labels['P_t'], labels['nu'], labels['P_nu'] )
```



1.2.2 Domain

Note: It is assumed that pyofss has been imported using

```
>>> from pyofss import *
```

The start of any system in pyofss is the domain. Generate a domain using

```
>>> domain = Domain()
```

which will construct a default domain. Information on the domain may be viewed

```
>>> print domain
Domain:
        total_bits = 1
        samples_per_bit = 512
        bit_width = 100.0000 ps
        centre_nu = 193.1000 THz
        total_samples = 512
        window_t = 100.0000 ps
        centre_omega = 1213.2831 rad / ps
        centre_lambda = 1552.5244 nm
        dt = 0.1953 ps
        window_nu = 5.1200 THz
        dnu = 0.0100 THz
        window_omega = 32.1699 rad / ps
        domega = 0.0628 rad / ps
        window_lambda = 41.1648 nm
        dlambda = 0.0804 nm
        channels = 1
```

The default domain uses a centre frequency of 193.1 THz, though often a centre wavelength of 1550 nm is used. There are helper functions to convert between frequency and wavelength. To find the frequency corresponding to the desired wavelength:

```
>>> nu_c = lambda_to_nu( 1550.0 )
>>> print nu_c
193.414489032
```

Now a domain may be constructed with this centre wavelength

```
>>> domain = Domain( centre_nu = nu_c )
>>> print domain
Domain:
        total_bits = 1
        samples_per_bit = 512
        bit_width = 100.0000 ps
        centre_nu = 193.4145 THz
        total_samples = 512
        window_t = 100.0000 ps
        centre_omega = 1215.2591 rad / ps
        centre_lambda = 1550.0000 nm
        dt = 0.1953 ps
        window_nu = 5.1200 THz
        dnu = 0.0100 THz
        window_omega = 32.1699 rad / ps
        domega = 0.0628 rad / ps
        window_lambda = 41.0311 nm
        dlambda = 0.0801 nm
        channels = 1
```

The temporal and spectral arrays may be accessed using

```
>>> domain.t
>>> domain.nu
>>> domain.Lambda
```

1.2.3 Gaussian

Note: It is assumed that pyofss has been imported using

```
>>> from pyofss import *
```

The Gaussian module allows a Gaussian shaped pulse to be generated.

A default Gaussian may be generated using:

```
>>> gaussian = Gaussian()
```

An example of a Gaussian with a peak power of 0.5 W and a width of 20 ps:

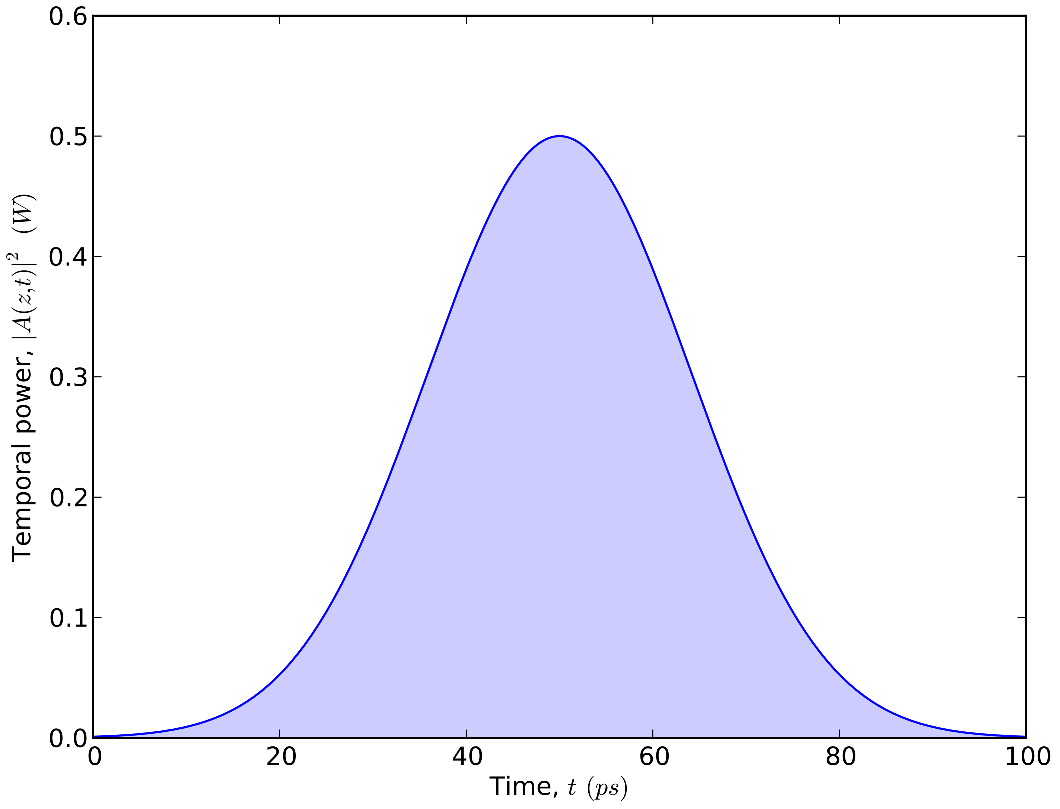
```
>>> gaussian = Gaussian( peak_power = 0.5, width = 20.0 )
```

The position parameter is relative to the temporal domain width (which may be found from sys.domain.window_t).

A simulation using a Gaussian pulse

```
from pyofss import *
sys = System()
sys.add( Gaussian(peak_power = 0.5, width = 20.0) )
sys.run()

single_plot( sys.domain.t, temporal_power(sys.field),
             labels['t'], labels['P_t'] )
```



1.2.4 Fibre

Note: It is assumed that pyofss has been imported using

```
>>> from pyofss import *
```

The fibre module may be regarded as the core of pyofss.

A default fibre may be generated using

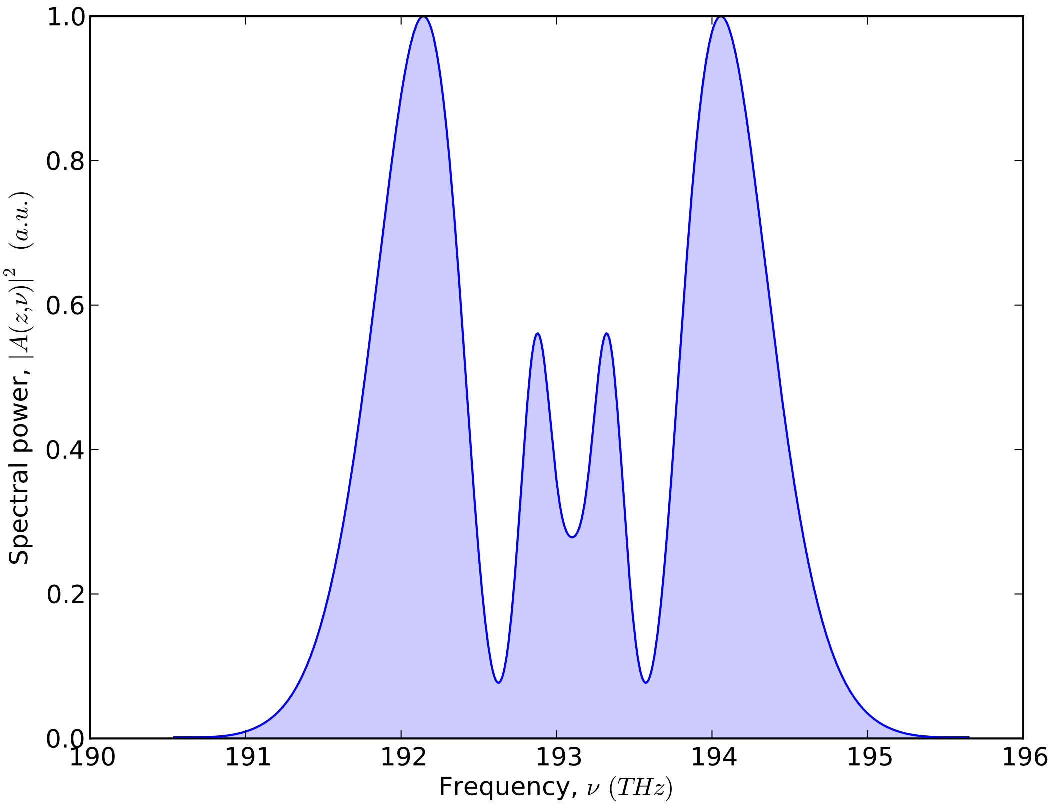
```
>>> fibre = Fibre()
```

For a more interesting fibre, modify the nonlinear parameter

```
>>> fibre = Fibre( gamma = 1.0 )
```

With an input pulse with sufficient power, there should be spectral broadening

```
from pyofss import *
sys = System()
sys.add( Gaussian(peak_power = 10.0, width = 1.0) )
sys.add( Fibre(gamma = 1.0) )
sys.run()
single_plot( sys.domain.nu, spectral_power(sys.field, True),
             labels['nu'], labels['P_nu'] )
```

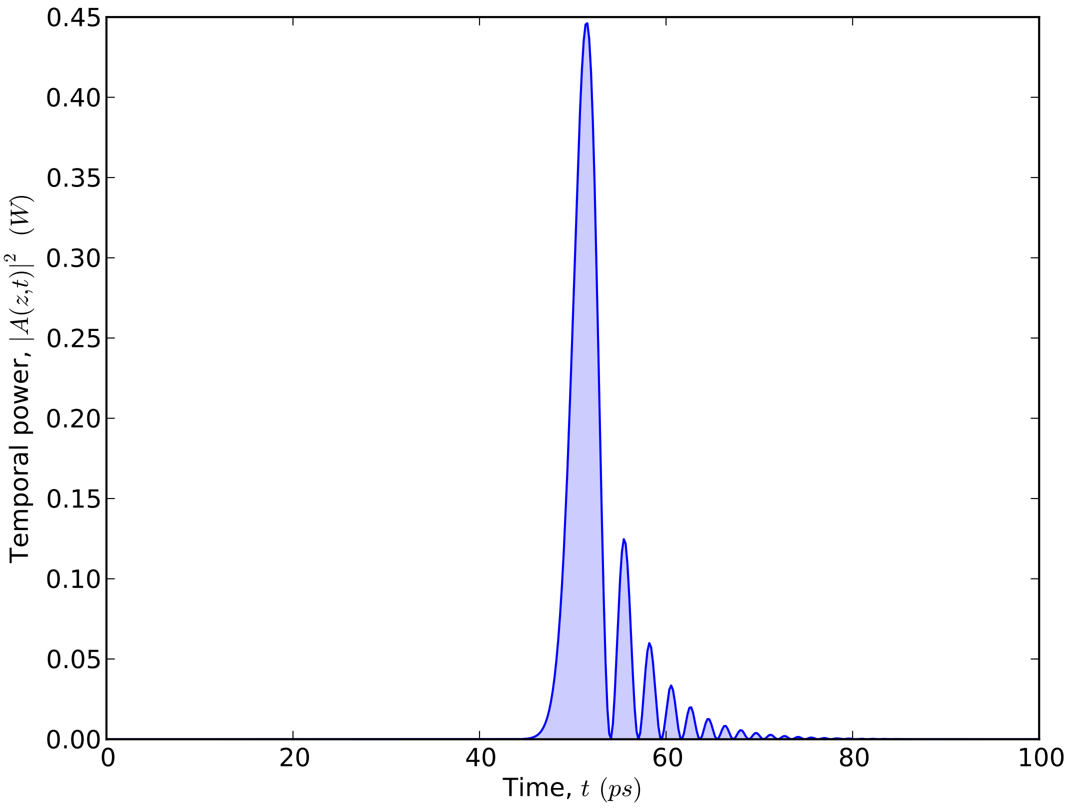


General dispersion may be included using the beta parameter. For third-order dispersion pass an array with third element (including a zeroth element) non-zero

```
>>> fibre = Fibre( beta = [0.0, 0.0, 0.0, 1.0] )
```

With an appropriate length of fibre the dispersion effects may be seen

```
from pyofss import *
sys = System()
sys.add( Gaussian(peak_power = 1.0, width = 1.0) )
sys.add( Fibre(beta = [0.0, 0.0, 0.0, 1.0], length = 10.0) )
sys.run()
single_plot( sys.domain.t, temporal_power(sys.field),
             labels['t'], labels['P_t'] )
```



1.3 Overview

1.3.1 Domain

A domain in pyofss contains the temporal and spectral axis (arrays) which are used by the system modules during simulation. There are helper functions which convert between frequency, angular frequency, and wavelength.

- TB – total bits
- SPB – samples per bit
- BW – bit width
- TS – total samples (TS = TB x SPB)
- TW – time window (TW = TB x BW)

Using $\Delta t_{max} \equiv TW$ and $N \equiv TS$, the temporal array corresponding to $t \in [0, \Delta t_{max})$ is generated from

$$\Delta t = \Delta t_{max}/N$$
$$t_n = n\Delta t \qquad n \in [0, N)$$

The first and last elements of the temporal array will have the values:

$$\begin{aligned} t[0] &= 0.0 \\ t[N-1] &= \Delta t_{max} - \Delta t \end{aligned}$$

For the spectral array there will be a fixed shift such that the middle of the array corresponds to a central frequency ν_c. Then the spectral array corresponding to $\nu \in \left[\nu_c - \frac{\Delta\nu_{max}}{2}, \nu_c + \frac{\Delta\nu_{max}}{2}\right)$ is generated from

$$\begin{aligned} \Delta\nu &= \frac{1}{\Delta t_{max}} \\ \Delta\nu_{max} &= N\Delta\nu \\ \nu_n &= \nu_c - \frac{\Delta\nu_{max}}{2} + n\Delta\nu \qquad n \in [0, N) \end{aligned}$$

The first and last elements of the spectral array will have the values:

$$\begin{aligned} \nu[0] &= \nu_c - \frac{\Delta\nu_{max}}{2} \\ \nu[N-1] &= \nu_c + \frac{\Delta\nu_{max}}{2} - \Delta\nu \end{aligned}$$

An angular frequency array and a wavelength array are also generated using the following two relations

$$\begin{aligned} \omega &= 2\pi\nu \\ \lambda &= c/\nu \end{aligned}$$

1.3.2 Field

The field in pyofss represents the complex valued slowly varying envelope of the electric field that will be propagated through the optical fibre system. A fixed polarisation is used for the field. It is possible to use separate field arrays for each frequency/wavelength (channel) to be simulated.

1.3.3 Modules

A module in pyofss is a class that may be called with a domain and a field as parameters. The following modules may be generated:

Gaussian

$$A(0,t) = \sqrt{P_0}\,\exp\left[\frac{-(1+iC)}{2}\left(\frac{t-t_0}{\Delta t_{1/e}}\right)^{2m} + i\left(\phi_0 - \ 2\pi(\nu_c + \nu_{\text{offset}})t\ \right)\right]$$

The gaussian module will generate a (super-)Gaussian shaped pulse using the following parameters:

P_0 – Peak power (W)

C – Chirp parameter (rad)

t_0 – pulse position (ps)

$\Delta t_{1/e}$ – pulse HWIeM width (ps) [**note: not FWHM**]

m – Order paramater

ϕ_0 – initial phase (rad)

ν_c – centre frequency (THz)

ν_{offset} – offset frequency (THz)

Since the centre frequency ν_c is fixed by the domain, only the offset frequency ν_{offset} is provided by the user.

The pulse position t_0 is given as a factor of the time window Δt_{max} such that $t_0 = f \Delta t_{max}$, and the user supplies the factor f.

Sech

$$A(0,t) = \sqrt{P_0}\,\text{sech}\left(\frac{t-t_0}{\Delta t_{1/e}}\right)\ \exp\left[\frac{-iC}{2}\left(\frac{t-t_0}{\Delta t_{1/e}}\right)^2 + i\,(\phi_0 - \ 2\pi(\nu_c + \nu_{\text{offset}})t\ )\right]$$

The sech module will generate a hyperbolic-Secant shaped pulse using the following parameters:

P_0 – Peak power (W)

C – Chirp parameter (rad)

t_0 – pulse position (ps)

$\Delta t_{1/e}$ – pulse HWIeM width (ps) [**note: not FWHM**]

ϕ_0 – initial phase (rad)

ν_c – centre frequency (THz)

ν_{offset} – offset frequency (THz)

Since the centre frequency ν_c is fixed by the domain, only the offset frequency ν_{offset} is provided by the user.

The pulse position t_0 is given as a factor of the time window Δt_{max} such that $t_0 = f \Delta t_{max}$, and the user supplies the factor f.

Generator

A generator module can generate Gaussian or hyperbolic-secant (Sech) shaped pulses. The main difference to directly calling a Gaussian or Sech module is that the pulse position paramter t_0 is given as a factor of the bit width, and not as a factor of the time window. The user provides the factor f where $t_0 = f \Delta t_{bit}$.

Fibre

The fibre module propagates the input field incrementally using the generalised non-linear Schrödinger equation:

$$\frac{\partial A}{\partial z} = \left[\hat{L} + \hat{N}\left(A\right)\right] A$$

where A is the complex field envelope of the pulse and z is the dimension along the fibre length. The linear operator $\hat{L}$ and non-linear operator $\hat{N}$ are usually written:

$$\hat{L} = -\frac{\alpha}{2} - \frac{i\beta_2}{2}\frac{\partial^2}{\partial t^2} + \frac{\beta_3}{6}\frac{\partial^3}{\partial t^3} + \dots$$

$$\hat{N} = i\gamma\left(|A|^2 + \frac{i}{\omega_0}\frac{1}{A}\frac{\partial |A|^2 A}{\partial t} - t_R\frac{\partial |A|^2}{\partial t}\right)$$

The linear operator contains terms for attenuation and (second order and higher) dispersion. The non-linear operator contains terms for self-phase modulation (SPM), self-steepening, and Raman scattering.

It is useful to apply the linear operator to the field in the frequency domain using the property $\frac{\partial A}{\partial t} \leftrightarrow -i\omega\tilde{A}$:

$$\hat{L} = -\frac{\alpha}{2} + i\left(\frac{\beta_2}{2}\omega^2 + \frac{\beta_3}{6}\omega^3 + \dots\right)$$

Filter

$$H(\nu) = \exp\left[\left(\frac{-2\pi\,\nu_{\text{filter}}}{2\Delta\nu}\right)^{2m}\right]$$

The filter module will slice a specified band of the current field in the spectral domain.

Amplifier

$$\tilde{A}_{\text{out}} = \sqrt{G}\tilde{A}_{\text{in}}$$

The amplifier module applies a gain to the input field.

1.4 Complete reference

1.4.1 Domain

class `pyofss.domain.`**`Domain`**(*total_bits=1*, *samples_per_bit=512*, *bit_width=100.0*, *centre_nu=193.09999999999999*, *channels=1*)

Parameters

- **total_bits** (*Uint*) – Total number of bits to generate
- **samples_per_bit** (*Uint*) – Number of samples to represent a bit
- **bit_width** (*double*) – Width of each bit. *Unit: ps*
- **centre_nu** (*double*) – Centre frequency. *Unit: THz*
- **channels** (*Uint*) – Number of channels to simulate. Used for WDM simulations

A domain consists of:

Bit data: total_bits, bit_width

Samples data: samples_per_bit, total_samples

Window size in each domain: window_t, window_nu, window_omega, window_lambda

Increment of each domain between two adjacent samples: dt, dnu, domega, dlambda

The generated domains: t, nu, omega, Lambda

Centre of spectral domains: centre_nu, centre_omega, centre_lambda

---

Note: Use of *Lambda*, and NOT the Python reserved word *lambda*

`__str__`()

Returns Information string

Return type string

Output information on Domain.

`vacuum_light_speed` = **299792.45799999998**

Speed of light in a vacuum. *Unit: nm / ps*

`pyofss.domain.`**`nu_to_omega`**(*nu*)

Parameters **nu** (*double*) – Frequency to convert. *Unit: THz*

Returns Angular frequency. *Unit: rad / ps*

Return type double

Convert from frequency to angular frequency

`pyofss.domain.`**`nu_to_lambda`**(*nu*)

Parameters **nu** (*double*) – Frequency to convert. *Unit: THz*

Returns Wavelength. *Unit: nm*

Return type double

Convert from frequency to wavelength

`pyofss.domain.`**`omega_to_nu`**`(`*omega*`)`

Parameters **omega** (*double*) – Angular frequency to convert. *Unit: rad / ps*

Returns Frequency. *Unit: THz*

Return type double

Convert from angular frequency to frequency

`pyofss.domain.`**`omega_to_lambda`**`(`*omega*`)`

Parameters **omega** (*double*) – Angular frequency to convert. *Unit: rad / ps*

Returns Wavelength. *Unit: nm*

Return type double

Convert from angular frequency to wavelength

`pyofss.domain.`**`lambda_to_nu`**`(`*Lambda*`)`

Parameters **Lambda** (*double*) – Wavelength to convert. *Unit: nm*

Returns Frequency. *Unit: THz*

Return type double

Convert from wavelength to frequency

`pyofss.domain.`**`lambda_to_omega`**`(`*Lambda*`)`

Parameters **Lambda** (*double*) – Wavelength to convert. *Unit: nm*

Returns Angular frequency. *Unit: rad / ps*

Return type double

Convert from wavelength to angular frequency

`pyofss.domain.`**`dnu_to_dlambda`**`(`*dnu*, *nu=193.09999999999999*`)`

Parameters

- **dnu** (*double*) – Small increment in frequency to convert. *Unit: THz*
- **nu** (*double*) – Reference frequency. *Unit: THz*

Returns Small increment in wavelength. *Unit: nm*

Return type double

Convert a small change in frequency to a small change in wavelength, with reference to centre frequency

`pyofss.domain.`**`dlambda_to_dnu`**`(`*dlambda*, *Lambda=1550.0*`)`

Parameters

- **dlambda** (*double*) – Small increment in wavelength to convert. *Unit: nm*
- **Lambda** (*double*) – Reference wavelength. *Unit: nm*

Returns Small increment in frequency. *Unit: THz*

Return type double

Convert a small change in wavelength to a small change in frequency, with reference to a centre wavelength

1.4.2 Field

`pyofss.field.`**`temporal_power`**(*A_t*, *normalise=False*)

Parameters

- **A_t** (*array_like*) – Input field array in the temporal domain
- **normalise** (*bool*) – Normalise returned array to that of maximum value

Returns Array of power values

Return type array_like

Generate an array of temporal power values from complex amplitudes array.

`pyofss.field.`**`spectral_power`**(*A_t*, *normalise=False*)

Parameters

- **A_t** (*array_like*) – Input field array in the temporal domain
- **normalise** (*bool*) – Normalise returned array to that of maximum value

Returns Array of power values

Return type array_like

Generate an array of spectral power values from complex amplitudes array. *Note: Expect input field to be in temporal domain. Never input A_nu!*

`pyofss.`**`phase`**(*A_t*, *unwrap=True*)

Parameters

- **A_t** (*array_like*) – Input field array in the temporal domain
- **unwrap** (*bool*) – Whether to unwrap phase angles from fixed range

Returns Array of phase angle values

Return type array_like

Generate an array of phase angles from complex amplitudes array.

`pyofss.`**`chirp`**(*A_t*, *window_nu*, *unwrap=True*)

Parameters

- **A_t** (*array_like*) – Input field array in the temporal domain
- **window_nu** (*double*) – Spectral window of the simulation
- **unwrap** (*bool*) – Whether to unwrap phase angles from fixed range

Returns Array of chirp values

Return type array_like

Generate an array of chirp values from complex amplitudes array.

`pyofss.`**`fft`**(*A_t*)

> **Parameters** **A_t** (*array_like*) – Input field array in the temporal domain
>
> **Returns** Output field array in the spectral domain
>
> **Return type** array_like

Fourier transform field from temporal domain to spectral domain. *Note: Physics convention – positive sign in exponential term.*

`pyofss.`**`ifft`**(*A_nu*)

> **Parameters** **A_nu** (*array_like*) – Input field array in the spectral domain
>
> **Returns** Output field array in the temporal domain
>
> **Return type** array_like

Inverse Fourier transform field from spectral domain to temporal domain. *Note: Physics convention – negative sign in exponential term.*

`pyofss.`**`ifftshift`**(*A_nu*)

> **Parameters** **A_nu** (*array_like*) – Input field array in the spectral domain
>
> **Returns** Shifted field array in the spectral domain
>
> **Return type** array_like

Shift the field values from "FFT order" to "consecutive order".

`pyofss.`**`fftshift`**(*A_nu*)

> **Parameters** **A_nu** (*array_like*) – Input field array in the spectral domain
>
> **Returns** Shifted field array in the spectral domain
>
> **Return type** array_like

Shift the field values from "consecutive order" to "FFT order".

1.4.3 Metrics

class `pyofss.metrics.`**`Metrics`**(*domain=None*, *field=None*)

Calculate useful metrics using a domain and a field. An example metric is Q.

`calculate`()

`pyofss.metrics.`**`calculate_regenerator_factor`**(*alpha*, *D*, *gamma*, *length*, *peak_power*, *using_alpha_dB=False*)

> **Parameters**
>
> - **alpha** (*double*) – Attenuation factor
> - **D** (*double*) – Dispersion
> - **length** (*double*) – Fibre length
> - **peak_power** (*double*) – Pulse peak power
> - **using_alpha_dB** (*bool*) – Whether using a logarithmic attnuation factor

Returns Maximum nonlinear phase and a factor indicating regeneration

Return type double, double

1.4.4 System

class `pyofss.system.`**`System`**(*domain=<pyofss.domain.Domain instance at 0x45018c0>*)

Parameters **domain** (*object*) – A domain to be used with contained modules

A system consists of a list of modules, each of which may be called with a domain and field as parameters. The result of each module call is stored in a dictionary.

`add`(*module*)

`clear`(*remove_modules=False*)
Clear contents of all fields. Clear (remove) all modules if requested.

`run`()

1.4.5 Amplifier

class `pyofss.modules.amplifier.`**`Amplifier`**(*name='amplifier'*, *gain=None*, *power=None*)

Parameters

- **name** (*string*) – Name of this module
- **gain** (*double*) – Amount of (logarithmic) gain. *Unit: dB*
- **power** (*double*) – Average power level to target

Simple amplifier provides gain but no noise

1.4.6 Bit

class `pyofss.modules.bit.`**`Bit`**(*position=0.5*, *width=10.0*, *peak_power=0.001*, *offset_nu=0.0*, *m=1*, *C=0.0*, *initial_phase=0.0*, *using_fwhm=False*)

Parameters

- **position** (*double*) – Position of pulse
- **width** (*double*) – Width of pulse
- **peak_power** (*double*) – Peak power of pulse
- **offset_nu** (*double*) – Offset frequency of pulse
- **m** (*Uint*) – Order parameter
- **C** (*double*) – Chirp parameter
- **initial_phase** (*double*) – Initial phase of the pulse
- **using_fwhm** (*bool*) – Determines whether the width parameter is a full- width at half maximum measure, or a half-width at 1/e maximum measure

Each bit is represented by a pulse.

`pyofss.modules.bit.`**`generate_bitstream`**`()`

class `pyofss.modules.bit.`**`Bit_stream`**

A bit_stream consists of a number of bits.

`add``(`*bit*`)`

`pyofss.modules.bit.`**`generate_prbs`**`(`*domain=None*, *bit=<pyofss.modules.bit.Bit instance at 0x45b7f38>*, *amplitude_jitter=0.0*, *ghost_pulse=0.0*`)`

Parameters

- **domain** (*object*) – Domain to use for calculations
- **bit** (*object*) – Bit parameters to use for generating a pulse
- **amplitude_jitter** (*double*) – Amount of amplitude jitter to add to pulse
- **ghost_pulse** (*double*) – Relative power for ghost pulses

Returns A bitstream with paramaters to generate pulses

Return type object

•**amplitude_jitter: percentage of variation in peak power of pulse** representing bit.

•**ghost_pulse: maximum peak power of a pulse representing a zero bit, as a** percentage of the mean peak power of pulses representing ones.

1.4.7 CW

class `pyofss.modules.cw.`**`Cw`**`(`*name='cw'*, *peak_power=0.0*, *offset_nu=0.0*, *initial_phase=0.0*, *channel=0*`)`

Parameters

- **name** (*string*) – Name of this module
- **peak_power** (*double*) – Peak power of the CW source
- **offset_nu** (*double*) – Offset frequency
- **initial_phase** (*double*) – Initial phase
- **channel** (*Uint*) – Channel sets the field to be modified

Generate a continuous wave (CW) source. Add this to appropriate field.

1.4.8 Attenuation

class `pyofss.modules.attenuation.`**`Attenuation`**`(`*alpha=None*, *sim_type=None*, *use_cache=False*`)`

Parameters

- **alpha** (*double*) – Attenuation factor
- **sim_type** (*string*) – Type of simulation, “default” or “wdm”

- **use_cache** (*bool*) – Cache exponential calculation optimisation

Attenuation is used by fibre to generate an attenuation array. Deprecated since version 0.8: This class has been subsumed into the linearity class. This function may be removed before version 1.0 is released.

default_attenuation(*domain*)

default_exp_f(*A*, *h*)

default_exp_f_cached(*A*, *h*)

default_f(*A*, *z*)

wdm_attenuation(*domain*)

wdm_exp_f(*As*, *h*)

wdm_exp_f_cached(*As*, *h*)

wdm_f(*As*, *z*)

pyofss.modules.attenuation.**convert_alpha_to_linear**(*alpha_dB=0.0*)

Parameters **alpha_dB** (*double*) – Logarithmic attenuation factor

Returns Linear attenuation factor

Return type double

Converts a logarithmic attenuation factor to a linear attenuation factor Deprecated since version 0.8: Use `convert_alpha_to_linear()` from linearity instead. This function may be removed before version 1.0 is released.

pyofss.modules.attenuation.**convert_alpha_to_dB**(*alpha_linear=0.0*)

Parameters **alpha_linear** (*double*) – Linear attenuation factor

Returns Logarithmic attenuation factor

Return type double

Converts a linear attenuation factor to a logarithmic attenuation factor Deprecated since version 0.8: Use `convert_alpha_to_dB()` from linearity instead. This function may be removed before version 1.0 is released.

1.4.9 Dispersion

class pyofss.modules.dispersion.**Dispersion**(*beta=None*, *sim_type=None*, *use_cache=False*, *centre_omega=None*)

Parameters

- **beta** (*array_like*) – Array of dispersion parameters
- **sim_type** (*string*) – Type of simulation, “default” or “wdm”
- **use_cache** (*bool*) – Cache calculated values if using fixed step-size
- **centre_omega** (*double*) – Angular frequency to use for dispersion array

Dispersion is used by fibre to generate a fairly general dispersion array. Deprecated since version 0.8: This class has been subsumed into the linearity class. This function may be removed before version 1.0 is released.

`default_dispersion`(*domain*)

`default_exp_f`(*A*, *h*)

`default_exp_f_cached`(*A*, *h*)

`default_f`(*A*, *z*)

`wdm_dispersion`(*domain*)

`wdm_exp_f`(*As*, *h*)

`wdm_exp_f_cached`(*As*, *h*)

`wdm_f`(*As*, *z*)

`pyofss.modules.dispersion.`**`convert_dispersion_to_physical`**(*D=0.0*, *S=0.0*, *Lambda=1550.0*)

Parameters

- **D** (*double*) – Dispersion. *Unit:* $ps/(nm \cdot km)$
- **S** (*double*) – Dispersion slope. *Unit:* $ps/(nm^2 \cdot km)$
- **Lambda** (*double*) – Centre wavelength. *Unit: nm*

Returns Second and third order dispersion

Return type double, double

Require D and S. Return a tuple containing beta_2 and beta_3. Deprecated since version 0.8: Use `convert_dispersion_to_physical()` from linearity instead. This function may be removed before version 1.0 is released.

`pyofss.modules.dispersion.`**`convert_dispersion_to_engineering`**(*beta_2=0.0*, *beta_3=0.0*, *Lambda=1550.0*)

Parameters

- **beta_2** (*double*) – Second-order dispersion. *Unit:* ps^2/km
- **beta_3** (*double*) – Third-order dispersion. *Unit:* ps^3/km
- **Lambda** (*double*) – Centre wavelength. *Unit: nm*

Returns Dispersion, dispersion slope

Return type double, double

Require beta_2 and beta_3. Return a tuple containing D and S. Deprecated since version 0.8: Use `convert_dispersion_to_engineering()` from linearity instead. This function may be removed before version 1.0 is released.

1.4.10 Linearity

class `pyofss.modules.linearity.`**`Linearity`**(*alpha=None*, *beta=None*, *sim_type=None*, *use_cache=False*, *centre_omega=None*)

Parameters

- **alpha** (*double*) – Attenuation factor
- **beta** (*array_like*) – Array of dispersion parameters
- **sim_type** (*string*) – Type of simulation, "default" or "wdm"
- **use_cache** (*bool*) – Cache calculated values if using fixed step-size
- **centre_omega** (*double*) – Angular frequency to use for dispersion array

Dispersion is used by fibre to generate a fairly general dispersion array.

`default_exp_f`(*A*, *h*)

`default_exp_f_cached`(*A*, *h*)

`default_f`(*A*, *z*)

`default_linearity`(*domain*)

`wdm_exp_f`(*As*, *h*)

`wdm_exp_f_cached`(*As*, *h*)

`wdm_f`(*As*, *z*)

`wdm_linearity`(*domain*)

`pyofss.modules.linearity.`**`convert_alpha_to_linear`**(*alpha_dB=0.0*)

Parameters alpha_dB (*double*) – Logarithmic attenuation factor

Returns Linear attenuation factor

Return type double

Converts a logarithmic attenuation factor to a linear attenuation factor

`pyofss.modules.linearity.`**`convert_alpha_to_dB`**(*alpha_linear=0.0*)

Parameters alpha_linear (*double*) – Linear attenuation factor

Returns Logarithmic attenuation factor

Return type double

Converts a linear attenuation factor to a logarithmic attenuation factor

`pyofss.modules.linearity.`**`convert_dispersion_to_physical`**(*D=0.0*, *S=0.0*, *Lambda=1550.0*)

Parameters

- **D** (*double*) – Dispersion. *Unit:* $ps/(nm \cdot km)$
- **S** (*double*) – Dispersion slope. *Unit:* $ps/(nm^2 \cdot km)$
- **Lambda** (*double*) – Centre wavelength. *Unit: nm*

Returns Second and third order dispersion

Return type double, double

Require D and S. Return a tuple containing beta_2 and beta_3.

`pyofss.modules.linearity.`**`convert_dispersion_to_engineering`**(*beta_2=0.0*, *beta_3=0.0*, *Lambda=1550.0*)

Parameters

- **beta_2** (*double*) – Second-order dispersion. *Unit:* ps^2/km
- **beta_3** (*double*) – Third-order dispersion. *Unit:* ps^3/km
- **Lambda** (*double*) – Centre wavelength. *Unit: nm*

Returns Dispersion, dispersion slope

Return type double, double

Require beta_2 and beta_3. Return a tuple containing D and S.

1.4.11 Fibre

class `pyofss.modules.fibre.`**`Fibre`**(*name='fibre'*, *length=1.0*, *alpha=None*, *beta=None*, *gamma=0.0*, *sim_type=None*, *traces=1*, *local_error=9.9999999999999995e-07*, *method='RK4IP'*, *total_steps=100*, *self_steepening=False*, *raman_scattering=False*, *rs_factor=0.0030000000000000001*, *use_all=False*, *centre_omega=None*, *tau_1=0.012200000000000001*, *tau_2=0.032000000000000001*, *f_R=0.17999999999999999*)

Parameters

- **name** (*string*) – Name of this module
- **length** (*double*) – Length of fibre
- **alpha** (*object*) – Attenuation of fibre
- **beta** (*object*) – Dispersion of fibre
- **gamma** (*double*) – Nonlinearity of fibre
- **sim_type** (*string*) – Type of simulation
- **traces** (*Uint*) – Number of field traces required
- **local_error** (*double*) – Relative local error used in adaptive stepper
- **method** (*string*) – Method to use in ODE solver
- **total_steps** (*Uint*) – Number of steps to use for ODE integration
- **self_steepening** (*bool*) – Toggles inclusion of self-steepening effects
- **raman_scattering** (*bool*) – Toggles inclusion of raman-scattering effects

- **rs_factor** (*float*) – Factor determining the amount of raman-scattering
- **use_all** (*bool*) – Toggles use of general expression for nonlinearity
- **centre_omega** (*double*) – Angular frequency used within dispersion class
- **tau_1** (*double*) – Constant used in Raman scattering calculation
- **tau_2** (*double*) – Constant used in Raman scattering calculation
- **f_R** (*double*) – Constant setting the fraction of Raman scattering used

sim_type is either default or wdm.

traces: If greater than 1, will save the field at uniformly-spaced points during fibre propagation. If zero, will output all saved points used. This is useful if using an adaptive stepper which will likely save points non-uniformly.

method: simulation method such as RK4IP, ARK4IP.

total_steps: If a non-adaptive stepper is used, this will be used to set the step-size between successive points along the fibre.

local_error: Relative local error to aim for between propagtion points.

`l` (*A*, *z*)

`linear` (*A*, *h*)

`n` (*A*, *z*)

`nonlinear` (*A*, *h*, *B*)

1.4.12 Filter

class `pyofss.modules.filter.`**`Filter`**(*name='filter'*, *width_nu=0.10000000000000001*, *nu=193.09999999999999*, *m=1*, *is_fwhm=False*)

Parameters

- **name** (*string*) – Name of this module
- **width_nu** (*double*) – Frequency bandwidth
- **nu** (*double*) – Frequency at which to filter
- **m** (*Uint*) – Order parameter
- **is_fwhm** (*bool*) – Decides if width_nu is FWHM or HWIeM

Generate a Gaussian filter.

1.4.13 Gaussian

class `pyofss.modules.gaussian.`**`Gaussian`**(*name='gaussian'*, *position=0.5*, *width=10.0*, *peak_power=0.001*, *offset_nu=0.0*, *m=1*, *C=0.0*, *initial_phase=0.0*, *channel=0*)

Parameters

- **name** (*string*) – Name of this module
- **position** (*double*) – Relative position within time window
- **width** (*double*) – Half width at 1/e of maximum. *Unit: ps*
- **peak_power** (*double*) – Power at maximum. *Unit: W*
- **offset_nu** (*double*) – Offset frequency relative to domain centre frequency. *Unit: THz*
- **m** (*Uint*) – Order parameter. m > 1 describes a super-Gaussian pulse
- **C** (*double*) – Chirp parameter. *Unit: rad*
- **initial_phase** (*double*) – Control the initial phase. *Unit: rad*
- **channel** (*Uint*) – Channel of the field array to modify if multi-channel.

`__call__`(*domain, field*)

Parameters

- **domain** (*object*) – A domain
- **field** (*object*) – Current field

Returns Field after modification by Gaussian

Return type Object

`__str__`()

Returns Information string

Return type string

Output information on Gaussian.

`generate`(*t*)

Parameters **t** (*Dvector*) – Temporal domain array

Returns Array of complex values. *Unit:* $\sqrt{W}$

Return type Cvector

Generate an array of complex values representing a Gaussian pulse.

1.4.14 Generator

class `pyofss.modules.generator.`**`Generator`**(*name='generator', bit_stream=None, channel=0*)

Parameters

- **name** (*string*) – Name of this module
- **bit_stream** (*object*) – An array of Bit objects
- **channel** (*Uint*) – Channel to modify

Generate a pulse with Gaussian or hyperbolic secant shape. Add this pulse to appropriate field, determined by channel.

`__call__`(*domain*, *field*)

Parameters

- **domain** (*object*) – A Domain
- **field** (*object*) – Current field

Returns Field after modification by Generator

Return type Object

1.4.15 Nonlinearity

class `pyofss.modules.nonlinearity.`**`Nonlinearity`**(*gamma=None*, *sim_type=None*, *self_steepening=False*, *raman_scattering=False*, *rs_factor=0.0030000000000000001*, *use_all=False*, *tau_1=0.012200000000000001*, *tau_2=0.032000000000000001*, *f_R=0.17999999999999999*)

Nonlinearity is used by fibre to generate a nonlinear factor.

`pyofss.modules.nonlinearity.`**`calculate_gamma`**(*nonlinear_index*, *effective_area*, *centre_omega=1213.2830828163781*)

Parameters

- **nonlinear_index** (*double*) – n_2. *Unit:* m^2/W
- **effective_area** (*double*) – A_{eff}. *Unit:* μm^2
- **centre_omega** (*double*) – Angular frequency of carrier. *Unit: rad / ps*

Returns Nonlinear parameter. γ. *Unit:* $rad/(Wkm)$

Return type double

Calculate nonlinear parameter from the nonlinear index and effective area.

`pyofss.modules.nonlinearity.`**`calculate_raman_term`**(*domain*, *tau_1=0.012200000000000001*, *tau_2=0.032000000000000001*)

Parameters

- **domain** (*object*) – A domain
- **tau_1** (*double*) – First adjustable parameter. *Unit: ps*
- **tau_2** (*double*) – Second adjustable parameter. *Unit: ps*

Returns Raman response function

Return type double

Calculate raman response function from tau_1 and tau_2.

1.4.16 Plotter

`pyofss.modules.plotter.`**`map_plot`**(*x*, *y*, *z*, *x_label=''*, *y_label=''*, *z_label=''*, *interpolation='lanczos'*, *use_colour=True*, *filename=''*, *dpi=100*, *y_range=None*)

Parameters

- **x** (*Dvector*) – First axis
- **y** (*Dvector*) – Second axis
- **z** (*Dvector*) – Third axis
- **x_label** (*string*) – Label for first axis
- **y_label** (*string*) – Label for second axis
- **z_label** (*string*) – Label for third axis
- **interpolation** (*string*) – Type of interpolation to use
- **use_colour** (*bool*) – Use colour plot (else black and white plot)
- **filename** (*string*) – Location to save file. If “”, use interactive plot
- **dpi** (*Uint*) – Resolution of output image (dots per inch)
- **y_range** (*Dvector*) – Change range of second axis if not None

Generate a map plot.

`pyofss.modules.plotter.`**`waterfall_plot`**(*x*, *y*, *z*, *x_label=''*, *y_label=''*, *z_label=''*, *use_poly=True*, *alpha=0.20000000000000001*, *filename=''*, *dpi=100*, *y_range=None*, *x_range=None*)

Parameters

- **x** (*Dvector*) – First axis
- **y** (*Dvector*) – Second axis
- **z** (*Dvector*) – Third axis
- **x_label** (*string*) – Label for first axis
- **y_label** (*string*) – Label for second axis
- **z_label** (*string*) – Label for third axis
- **use_poly** (*bool*) – Whether to use filled polygons
- **alpha** (*double*) – Set transparency of filled polygons
- **filename** (*string*) – Location to save file. If “”, use interactive plot
- **dpi** (*Uint*) – Resolution of output image (dots per inch)
- **y_range** (*Dvector*) – Change range of second axis if not None
- **x_range** (*Dvector*) – Change range of first axis if not None

Generate a waterfall plot.

`pyofss.modules.plotter.`**`single_plot`**(*x*, *y*, *x_label=''*, *y_label=''*, *label=''*, *x_range=None*, *y_range=None*, *use_fill=True*, *alpha=0.20000000000000001*, *filename=''*, *dpi=100*)

Parameters

- **x** (*Dvector*) – First axis
- **y** (*Dvector*) – Second axis
- **x_label** (*string*) – Label for first axis
- **y_label** (*string*) – Label for second axis
- **label** (*string*) – Label for plot area
- **x_range** (*Dvector*) – Change range of first axis if not None
- **y_range** (*Dvector*) – Change range of second axis if not None
- **use_fill** (*bool*) – Fill area between plot line and axis
- **alpha** (*double*) – Transparency of filled area
- **filename** (*string*) – Location to save file. If "", use interactive plot
- **dpi** (*Uint*) – Resolution of output image (dots per inch)

Generate a single plot.

`pyofss.modules.plotter.`**`double_plot`**(*x*, *y*, *X*, *Y*, *x_label=''*, *y_label=''*, *X_label=''*, *Y_label=''*, *x_range=None*, *y_range=None*, *X_range=None*, *Y_range=None*, *use_fill=True*, *alpha=0.20000000000000001*, *filename=''*, *dpi=100*)

Parameters

- **x** (*Dvector*) – First axis of upper plot
- **y** (*Dvector*) – Second axis of upper plot
- **X** (*Dvector*) – First axis of lower plot
- **Y** (*Dvector*) – Second axis of lower plot
- **x_label** (*string*) – Label for first axis of upper plot
- **y_label** (*string*) – Label for second axis of upper plot
- **X_label** (*string*) – Label for first axis of lower plot
- **Y_label** (*string*) – Label for second axis of lower plot
- **x_range** (*Dvector*) – Change first axis range of upper plot if not None
- **y_range** (*Dvector*) – Change second axis range of upper plot if not None
- **X_range** (*Dvector*) – Change first axis range of lower plot if not None
- **Y_range** (*Dvector*) – Change second axis range of lower plot if not None
- **use_fill** (*bool*) – Fill area between plot line and axis

- **alpha** (*double*) – Transparency of filled area
- **filename** (*string*) – Location to save file. If “”, use interactive plot
- **dpi** (*Uint*) – Resolution of output image (dots per inch)

Generate a double plot. The two plots will be arranged vertically, one above the other.

`pyofss.modules.plotter.`**`multi_plot`**(*x, ys, zs, x_label='‘, y_label='‘, z_labels=['‘], x_range=None, y_range=None, use_fill=True, alpha=0.20000000000000001, filename='‘, dpi=100*)

Parameters

- **x** (*Dvector*) – First axis
- **ys** (*VDvector*) – Array of second axis
- **zs** (*VDvector*) – Array of third axis
- **x_label** (*string*) – Label for first axis
- **y_label** (*string*) – Label for second axis
- **Vstring** – List of strings for third axis
- **x_range** (*Dvector*) – Change range of first axis if not None
- **y_range** (*Dvector*) – Change range of second axis if not None
- **use_fill** (*bool*) – Fill area between plot line and axis
- **alpha** (*double*) – Transparency of filled area
- **filename** (*string*) – Location to save file. If “”, use interactive plot
- **dpi** (*Uint*) – Resolution of output image (dots per inch)

Generate a multiple overlapping plots.

`pyofss.modules.plotter.`**`quad_plot`**(*x, ys, zs, x_label='‘, y_label='‘, z_labels=['‘], x_range=None, y_range=None, use_fill=True, alpha=0.20000000000000001, filename='‘, dpi=100*)

Parameters

- **x** (*Dvector*) – First axis
- **ys** (*VDvector*) – Array of second axis
- **zs** (*VDvector*) – Array of third axis
- **x_label** (*string*) – Label for first axis
- **y_label** (*string*) – Label for second axis
- **Vstring** – List of strings for third axis
- **x_range** (*Dvector*) – Change range of first axis if not None
- **y_range** (*Dvector*) – Change range of second axis if not None
- **use_fill** (*bool*) – Fill area between plot line and axis

- **alpha** (*double*) – Transparency of filled area
- **filename** (*string*) – Location to save file. If "", use interactive plot
- **dpi** (*Uint*) – Resolution of output image (dots per inch)

Generate four plots arranged in a two-by-two square.

`pyofss.modules.plotter.`**`animated_plot`** (*x*, *y*, *z*, *x_label='`*, *y_label='`*, *z_label='`*, *x_range=None*, *y_range=None*, *alpha=0.20000000000000001*, *fps=5*, *clear_temp=True*, *frame_prefix='_tmp'*, *filename='`*)

Parameters

- **x** (*Dvector*) – First axis
- **y** (*VDvector*) – Array of second axis
- **z** (*Dvector*) – Third axis
- **x_label** (*string*) – Label for first axis
- **y_label** (*string*) – Label for second axis
- **z_label** (*string*) – label for third axis
- **x_range** (*Dvector*) – Change range of first axis if not None
- **y_range** (*Dvector*) – Change range of second axis if not None
- **alpha** (*double*) – Transparency of filled area
- **fps** (*Uint*) – Number of frames to show every second (frames per second)
- **clear_temp** (*bool*) – Remove temporary image files after completion
- **frame_prefix** (*string*) – Prefix used for each temporary file
- **filename** (*string*) – Location to save file. If "" then interactive plot

Generate an animated plot, either interactive or saved as a video.

`pyofss.modules.plotter.`**`convert_video`** (*filename*, *output='ogv'*)

Parameters

- **filename** (*string*) – Location of movie file to convert
- **output** (*string*) – Convert movie to output video type

Convert video to ogv, webm, or mp4 type.

`pyofss.modules.plotter.`**`generate_frames`** (*x*, *y*, *z*, *x_label='`*, *y_label='`*, *z_label='`*, *x_range=None*, *y_range=None*, *filepath='dat/'*, *dpi=100*)

Parameters

- **x** (*Dvector*) – First axis
- **y** (*VDvector*) – Array of second axis
- **z** (*Dvector*) – Third axis

- **x_label** (*string*) – Label for first axis
- **y_label** (*string*) – Label for second axis
- **z_label** (*string*) – label for third axis
- **x_range** (*Dvector*) – Change range of first axis if not None
- **y_range** (*Dvector*) – Change range of second axis if not None
- **filepath** (*string*) – Location to store each frame (plot)
- **dpi** (*Uint*) – Resolution of output image (dots per inch)

Generate multiple plots, storing each in a directory given by filepath. Deprecated since version 0.7: Use `animated_plot()` instead. This function may be removed before version 1.0 is released.

`pyofss.modules.plotter.`**`generate_movie`**(*data_path='dat/'*, *width=640*, *height=480*, *fps=5*, *filename='output.avi'*)

Parameters

- **data_path** (*string*) – Location of directory containing frames (plots)
- **width** (*Uint*) – Width of movie
- **height** (*Uint*) – Height of movie
- **fps** (*Uint*) – Number of frames to show every second (frames per second)
- **filename** (*string*) – Output filename for movie

Stitch together multiple plots found in data_path directory to form a movie. Deprecated since version 0.7: Use `animated_plot()` instead. This function may be removed before version 1.0 is released.

`pyofss.modules.plotter.`**`convert_movie`**(*filename*, *output='ogv'*)

Parameters

- **filename** (*string*) – Location of movie file to convert
- **output** (*string*) – Convert movie to output video type

Convert video to ogv, webm, or mp4 type. Deprecated since version 0.7: Use `convert_video()` instead. This function may be removed before version 1.0 is released.

1.4.17 Sech

class `pyofss.modules.sech.`**`Sech`**(*name='sech'*, *position=0.5*, *width=10.0*, *peak_power=0.001*, *offset_nu=0.0*, *m=0*, *C=0.0*, *initial_phase=0.0*, *channel=0*)

Parameters

- **name** (*string*) – Name of this module
- **position** (*double*) – Relative position within time window
- **width** (*double*) – Half width at 1/e of maximum. *Unit: ps*

- **peak_power** (*double*) – Power at maximum. *Unit: W*
- **offset_nu** (*double*) – Offset frequency relative to domain centre frequency. *Unit: THz*
- **m** (*Uint*) – Order parameter. m = 0. **Unused**
- **C** (*double*) – Chirp parameter. *Unit: rad*
- **initial_phase** (*double*) – Control the initial phase. *Unit: rad*
- **channel** (*Uint*) – Channel of the field array to modify if multi-channel.

`__call__`(*domain, field*)

Parameters

- **domain** (*object*) – A domain
- **field** (*object*) – Current field

Returns Field after modification by Sech

Return type Object

`__str__`()

Returns Information string

Return type string

Output information on Sech.

`generate`(*t*)

Parameters **t** (*Dvector*) – Temporal domain array

Returns Array of complex values. *Unit:* $\sqrt{W}$

Return type Cvector

Generate an array of complex values representing a Sech pulse.

1.4.18 Solver

class `pyofss.modules.solver.`**`Solver`**(*method='rk4', f=None*)

Parameters

- **method** (*string*) – Name of solver method to be used
- **f** (*object*) – Derivative function

Solver consists of both explicit and embedded routines for ODE integration.

Note: For embedded methods: Even though the calculated error applies to A_coarse, it has become common to use a weighted combination of A_coarse and A_fine to produce a higher order result (local extrapolation). It is for this reason that the embedded methods only return A_fine rather than A_coarse, or even both (A_coarse, A_fine). To be strict, return A_coarse in the methods.

`__call__` (*A*, *z*, *h*)
Return A_fine, calculated by method.

`bs` (*A*, *z*, *h*, *f*)
Bogacki-Shampine method (local orders: three and two)

`ck` (*A*, *z*, *h*, *f*)
Cash-Karp method (local order: four and five)

`dp` (*A*, *z*, *h*, *f*)
Dormand-Prince method (local orders: four and five)

`euler` (*A*, *z*, *h*, *f*)
Euler method

`midpoint` (*A*, *z*, *h*, *f*)
Midpoint method

`rk4` (*A*, *z*, *h*, *f*)
Runge-Kutta fourth-order method

`rk4ip` (*A*, *z*, *h*, *f*)
Runge-Kutta in the interaction picture method

`rkf` (*A*, *z*, *h*, *f*)
Runge-Kutta-Fehlberg method (local orders: four and five)

`ss_agrawal` (*A*, *z*, *h*, *f*)
Agrawal (iterative) split-step method

`ss_reduced` (*A*, *z*, *h*, *f*)
Reduced split-step method

`ss_simple` (*A*, *z*, *h*, *f*)
Simple split-step method

`ss_sym_midpoint` (*A*, *z*, *h*, *f*)
Symmetric split-step method (midpoint method for nonlinear)

`ss_sym_rk4` (*A*, *z*, *h*, *f*)
Symmetric split-step method (classical Runge-Kutta for nonlinear)

`ss_symmetric` (*A*, *z*, *h*, *f*)
Symmetric split-step method

`ssfm` (*A*, *z*, *h*, *f*)
Deprecated since version 0.8: Use `ss_simple()` instead. This function may be removed before version 1.0 is released.

`ssfm_reduced` (*A*, *z*, *h*, *f*)
Deprecated since version 0.8: Use `ss_reduced()` instead. This function may be removed before version 1.0 is released.

`ssfm_sym` (*A*, *z*, *h*, *f*)
Deprecated since version 0.8: Use `ss_symmetric()` instead. This function may be removed before version 1.0 is released.

`ssfm_sym_midpoint` (*A*, *z*, *h*, *f*)
Deprecated since version 0.8: Use `ss_sym_midpoint()` instead. This function may be removed before version 1.0 is released.

ssfm_sym_rk4 (*A*, *z*, *h*, *f*)
: Deprecated since version 0.8: Use `ss_sym_rk4()` instead. This function may be removed before version 1.0 is released.

ssfm_sym_rkf (*A*, *z*, *h*, *f*)
: Deprecated since version 0.8: This function may be removed before version 1.0 is released.

1.4.19 Stepper

class `pyofss.modules.stepper.`**`Stepper`** (*traces=1*, *local_error=9.9999999999999995e-07*, *method='RK4'*, *f=None*, *length=1.0*, *total_steps=100*)

Parameters

- **traces** (*Uint*) – Number of ouput trace to use
- **local_error** (*double*) – Relative local error required in adaptive method
- **method** (*string*) – ODE solver method to use
- **f** (*object*) – Derivative function to be solved
- **length** (*double*) – Length to integrate over
- **total_steps** (*Uint*) – Number of steps to use for ODE integration

method:

- EULER – Euler method;
- MIDPOINT – Midpoint method;
- RK4 – Fourth order Runge-Kutta method;
- BS – Bogacki-Shampine method;
- RKF – Runge-Kutta-Fehlberg method;
- CK – Cash-Karp method;
- DP – Dormand-Prince method;
- SS_SIMPLE – Simple split-step method;
- SS_SYMMETRIC – Symmetric split-step method;
- SS_REDUCED – Reduced split-step method;
- SS_AGRAWAL – Agrawal (iterative) split-step method;
- SS_SYM_MIDPOINT – Symmetric split-step method (MIDPOINT for nonlinear)
- SS_SYM_RK4 – Symmetric split-step method (RK4 for nonlinear);
- RK4IP – Runge-Kutta in the interaction picture method.

Each method may use an adaptive stepper by prepending an 'A' to the name.

traces:

- 0 – Store A for each succesful step;
- 1 – Store A at final value (length) only;

- **>1 – Store A for each succesful step then use interpolation to get A** values for equally spaced z-values, calculated using traces.

`__call__`(*A*)
Delegate to appropriate function, adaptive- or standard-stepper

`adaptive_stepper`(*A*)
Take multiple steps, with variable length, until target reached

`relative_local_error`(*A_fine*, *A_coarse*)
Calculate an estimate of the relative local error

`standard_stepper`(*A*)
Take a fixed number of steps, each of equal length

1.4.20 Storage

class `pyofss.modules.stepper.`**`Storage`**
Contains A arrays for multiple z values. Also contains t array and functions to modify the stored data.

`append`(*z*, *A*)

Parameters

- **z** (*double*) – Distance along fibre
- **A** (*array_like*) – Field at distance z

Append current fibre distance and field to stored array

`find_nearest`(*array*, *value*)

Parameters

- **array** (*array_like*) – Array in which to locate value
- **value** (*double*) – Value to locate within array

Returns Index and element of array corresponding to value

Return type Uint, double

`get_plot_data`(*is_temporal=True*, *reduced_range=None*, *normalised=False*, *channel=None*)

Parameters

- **is_temporal** (*bool*) – Use temporal domain data (else spectral domain)
- **reduced_range** (*Dvector*) – Reduced x_range. Reduces y array to match.
- **normalised** (*bool*) – Normalise y array to first value.
- **channel** (*Uint*) – Channel number if using WDM simulation.

Returns Data for x, y, and z axis

Return type Tuple

Generate data suitable for plotting. Includes temporal/spectral axis, temporal/spectral power array, and array of z values for the x,y data.

`interpolate_As` (*zs*, *As*)

Parameters

- **zs** (*array_like*) – z values to find interpolated A
- **As** (*array_like*) – Array of As to be interpolated

Returns Interpolated As

Return type array_like

Interpolate array of A values, stored at non-uniform z-values, over a uniform array of new z-values (zs).

`interpolate_As_for_z_values` (*zs*)

Parameters **zs** (*array_like*) – Array of z values for which A is required

Split into separate arrays, interpolate each, then re-join.

`reduce_to_range` (*first_t*, *last_t*)

Parameters

- **first_t** (*double*) – Initial value of temporal range
- **last_t** (*double*) – Final value of temporal range

Reduce the size of stored arrays, As and t. Start by finding left and right index corresponding to first_t and last_t respectively. Use these indices to slice both t array and As array. Deprecated since version 0.8: Use non-member `reduce_to_range()` instead. This function may be removed before version 1.0 is released.

`reset_fft_counter` ()

Resets the global variable located in the field module.

`store_current_fft_count` ()

Store current value of the global variable in the field module.

`pyofss.modules.storage.reduce_to_range` (*x*, *ys*, *first_value*, *last_value*)

Parameters

- **x** (*array_like*) – Array of x values to search
- **ys** (*array_like*) – Array of y values corresponding to x array
- **first_value** – Initial value of required range
- **last_value** – Final value of required range

Returns Reduced x and y arrays

Return type array_like, array_like

From a range given by first_value and last_value, attempt to reduce x array to required range while also reducing corresponding y array.

CHAPTER

TWO

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

INDEX