SIMSSNR

Release 1.0.0

Valerii Brudanin (TU Delft)

Dec 11, 2024

CONTENTS:

CHAPTER

ONE

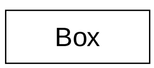
WELCOME TO SIM DOCUMENTATION

1.1 SIM

1.1.1 Box module

Box.py

This module contains classes for handling simulation volume and containing fields.



class Box.Box(*sources=()*, *box_size=10*, *point_number=100*, *additional_info=None*)

Bases: object

This class represents a simulation volume where fields and intensities are computed.

info

Additional information about the box.

Type
dict

box_size

Size of the box in each dimension.

Type
np.ndarray

point_number

Number of points in each dimension.

Type
np.ndarray

box_volume

Volume of the box.

Type
float

fields
    List of fields in the box.

    Type
        list

numerically_approximated_intensity_fields
    List of numerically approximated intensity fields.

    Type
        list

source_identifier
    Identifier for the sources.

    Type
        int

axes
    Axes for the box.

    Type
        tuple

frequency_axes
    Frequency axes for the box.

    Type
        tuple

grid
    Grid of points in the box.

    Type
        np.ndarray

electric_field
    Electric field in the box.

    Type
        np.ndarray

intensity
    Intensity in the box.

    Type
        np.ndarray

numerically_approximated_intensity
    Numerically approximated intensity in the box.

    Type
        np.ndarray

intensity_fourier_space
    Intensity in the Fourier space.

    Type
        np.ndarray

numerically_approximated_intensity_fourier_space

Numerically approximated intensity in the Fourier space.

Type

np.ndarray

analytic_frequencies

List of analytic frequencies.

Type

list

add_source(*source*)

Adds a source to the box. The corresponding field is added automatically.

Parameters

source – Source to add.

compute_axes()

Computes the axes and frequency axes for the box.

Returns

Axes and frequency axes for the box.

Return type

tuple

compute_electric_field()

Computes the electric field in the box.

compute_grid()

Computes the grid of points in the box.

compute_intensity_and_spatial_waves_numerically()

Find approximately spatial waves from intensity in Fourier space and compute from them the approximated intensity in the box.

compute_intensity_fourier_space()

compute_intensity_from_electric_field()

Computes the intensity from the electric field.

compute_intensity_from_spatial_waves()

Computes the intensity from intensity spatial waves.

get_approximated_intensity_sources()

Returns a list of numerically estimated intensity sources in the box.

Returns

List of approximated intensity sources.

Return type
list

get_plane_waves()

Returns a list of plane waves in the box.

Returns
List of plane waves.

Return type
list

get_sources()

Returns a list of sources in the box.

Returns
List of sources.

Return type
list

get_spatial_waves()

Returns a list of spatial waves in the box.

Returns
List of spatial waves.

Return type
list

plot_approximate_intensity_fourier_space_slices(*ax=None*, *slider=None*)

Plots slices of the intensity in the Fourier space, computed from spatial waves found numerically
.

Parameters

- ax (matplotlib.axes.Axes, optional) – Axes to plot on. Defaults to None.
- slider (matplotlib.widgets.Slider, optional) – Slider for interactive plotting. Defaults to None.

plot_approximate_intensity_slices(*ax=None*, *slider=None*)

Plots slices of the intensity in the real space, computed from spatial waves found numerically.

Parameters

- ax (matplotlib.axes.Axes, optional) – Axes to plot on. Defaults to None.
- slider (matplotlib.widgets.Slider, optional) – Slider for interactive plotting. Defaults to None.

plot_intensity_fourier_space_slices(*ax=None*, *slider=None*)

Plots slices of the intensity in the Fourier space.

Parameters

- ax (matplotlib.axes.Axes, optional) – Axes to plot on. Defaults to None.
- slider (matplotlib.widgets.Slider, optional) – Slider for interactive plotting. Defaults to None.

plot_intensity_slices(*ax=None*, *slider=None*)

Plots slices of the intensity in the real space.

Parameters

- ax (matplotlib.axes.Axes, optional) – Axes to plot on. Defaults to None.
- slider (matplotlib.widgets.Slider, optional) – Slider for interactive plotting. Defaults to None.

plot_slices(*array3d*, *ax=None*, *slider=None*)

Plots slices of a 3D array.

Parameters

- array3d (np.ndarray) – 3D array to plot.
- ax (matplotlib.axes.Axes, optional) – Axes to plot on. Defaults to None.
- slider (matplotlib.widgets.Slider, optional) – Slider for interactive plotting. Defaults to None.

remove_source(*source_identifier*)

Removes a source from the box by its identifier. The corresponding field is removed as well.

Parameters

source_identifier (int) – Identifier of the source to remove.

class Box.BoxSIM(*illumination: ~Illumination.Illumination = <Illumination.Illumination object>*, *box_size=10*, *point_number=100*, *additional_info=None*)

Bases: Box

This class is an extension of the class Box that supports SIM specific operations, such as illumination shifts.

illumination

The illumination configuration.

Type

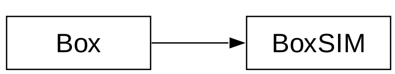
IlluminationConfiguration

illuminations_shifted

Array of shifted illuminations for different angles and shifts.

Type

np.ndarray



compute_total_illumination() → ndarray

get_intensity(*r: int*, *n: int*) → ndarray class Box.Field(*source:* Source, *grid: ndarray[tuple[int, int, int, 3], float64]*, *identifier: int*)

Bases: object

This class keeps field values within a given numeric volume.

identifier

Unique identifier for the field.

Type

int

field_type

Type of the field (either “ElectricField” or “Intensity”).

Type

str

source

The source that produces the field.

Type

Source

field

The computed field values.

Type

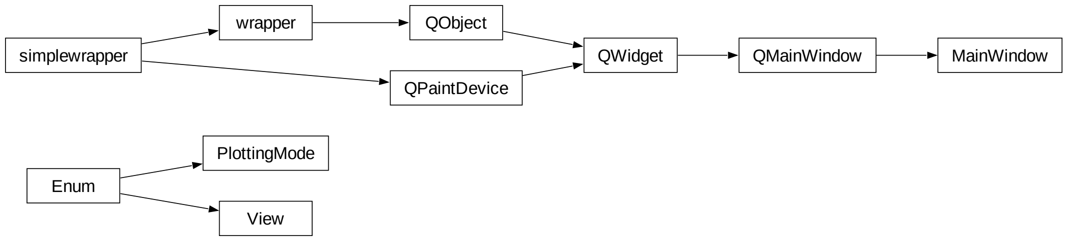
np.ndarray

1.1.2 GUI module

GUI.py

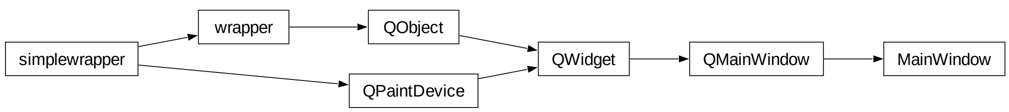
This module contains the main graphical user interface (GUI) components of the application.

This module and related ones is currently a demo-version of the user-interface, and will possibly be sufficiently modified or replaced in the future. For this reason, no in-depth documentation is provided.



class GUI.MainWindow(*box=None*)

Bases: QMainWindow



add_intensity_plane_wave(*ipw=None*)

add_plane_wave(*ipw=None*)

add_point_source()

add_source(*source*)

add_to_box(*initialized*, *source*)

change_plotting_mode()

change_view3d()

choose_plotting_mode(*Z*)

choose_view3d(*array*, *number*)

clear_layout(*layout*)

compute_and_plot_fourier_space()

compute_and_plot_from_electric_field()

compute_and_plot_from_intensity_sources()

compute_next_shift()

compute_numerically_approximated_intensities()

compute_total_intensity()

get_ipw_from_pw()

init_ui()

load_config()

load_illumination()

on_option_selected(*index*)

plot_fourier_space_slices(*intensity=None*)

plot_intensity_slices(*intensity=None*)

plot_numerically_approximated_intensity()

plot_numerically_approximated_intensity_fourier_space()

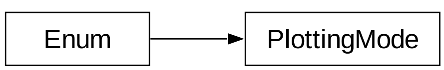
plot_shift_arrow()

remove_source(*initializer*)

save_config()

class GUI.PlottingMode(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum



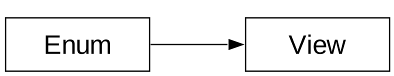
linear = 0

logarithmic = 1

mixed = 2

class GUI.View(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum



XY = 0

XZ = 2

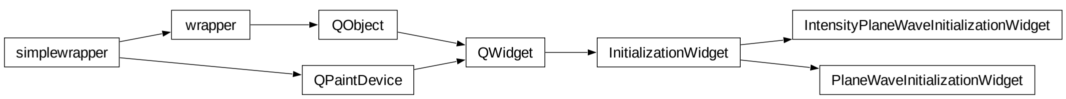
YZ = 1

1.1.3 GUIInitializationWidgets module

GUIInitializationWidgets.py

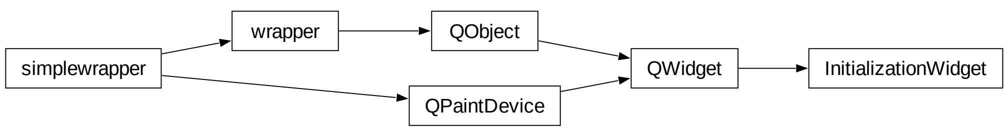
This module contains classes and functions for initializing GUI widgets.

This module and related ones is currently a demo-version of the user-interface, and will possibly be sufficiently modified or replaced in the future. For this reason, no in-depth documentation is provided.



class GUIInitializationWidgets.InitializationWidget

Bases: QWidget

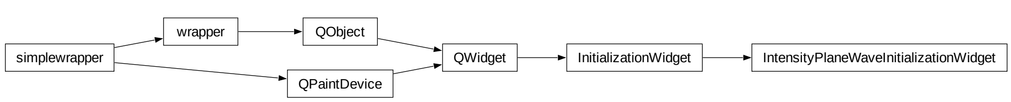


abstract on_click_ok()

abstract request_data()

class GUIInitializationWidgets.IntensityPlaneWaveInitializationWidget

Bases: InitializationWidget



on_click_ok()

request_data()

sendInfo

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

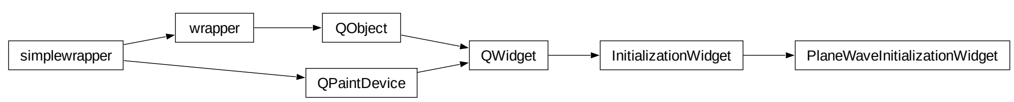
types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

class GUIInitializationWidgets.PlaneWaveInitializationWidget

Bases: InitializationWidget



on_click_ok()

request_data()

sendInfo

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

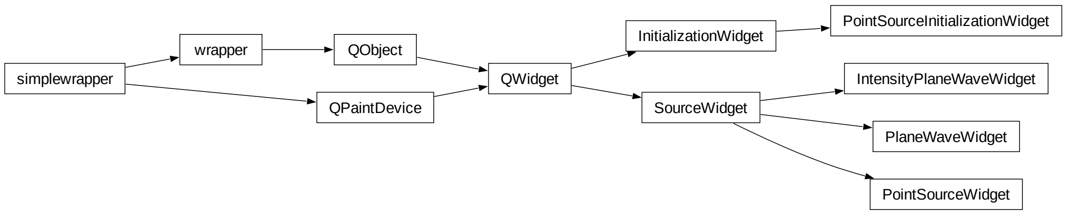
Type
str = ..., revision

1.1.4 GUIWidgets module

GUIWidgets.py

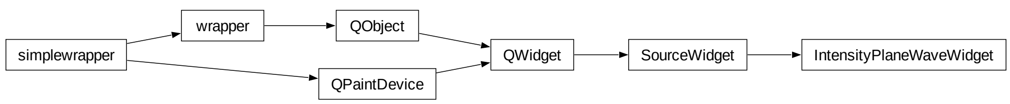
This module contains utility widgets for the GUI components.

This module and related ones is currently a demo-version of the user-interface, and will possibly be sufficiently modified or replaced in the future. For this reason, no in-depth documentation is provided.



class GUIWidgets.IntensityPlaneWaveWidget(*ipw*)

Bases: SourceWidget



change_widget()

contextMenuEvent(*self*, *a0: QContextMenuEvent | None*)

init_ui(*ipw*)

isSet

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

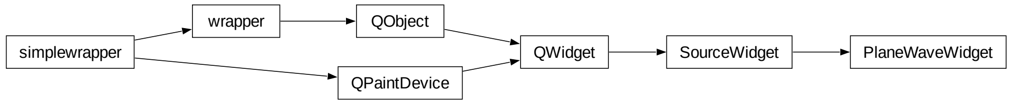
Type
pyqtSignal(*types, name

Type
str = ..., revision

on_receive_info(*info*)

class GUIWidgets.PlaneWaveWidget(*pw=None*)

Bases: SourceWidget



change_widget()

contextMenuEvent(*self*, *a0: QContextMenuEvent | None*)

init_ui(*pw*)

isDeleted

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

isSet

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

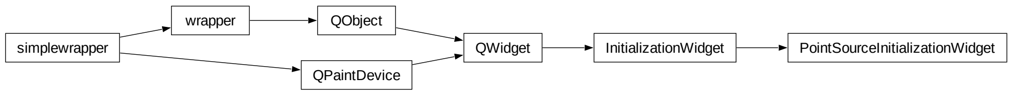
Type
pyqtSignal(*types, name

Type
str = ..., revision

on_receive_info(*info*)

class GUIWidgets.PointSourceInitializationWidget

Bases: InitializationWidget



non_numbers = ['', '-']

request_data()

sendBrightness

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

sendCoordinates

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

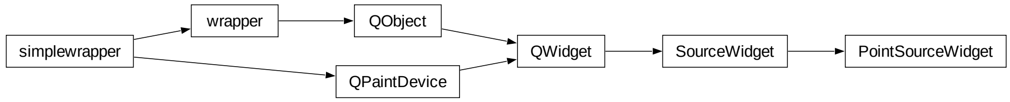
Type
str = ..., revision

send_brightness()

send_coordinates()

class GUIWidgets.PointSourceWidget

Bases: SourceWidget



change_widget()

contextMenuEvent(*self*, *a0: QContextMenuEvent | None*)

init_ui()

isSet

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

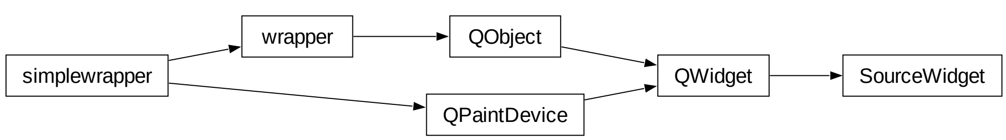
on_click_ok()

on_receive_brightness(*brightness*)

on_receive_coordinates(*coordinates*)

class GUIWidgets.SourceWidget(*source*)

Bases: QWidget



abstract change_widget()

abstract contextMenuEvent(*self*, *a0: QContextMenuEvent | None*)

identifier = 0

abstract init_ui(*source*)

isDeleted

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

remove_widget()

1.1.5 Illumination module

Illumination.py

This module contains the Illumination class, which handles the simulation and analysis of illumination patterns in optical systems.

Classes:
Illumination: Manages the properties and behavior of illumination patterns, including wavevectors and spatial shifts.

class Illumination.Illumination(*intensity_plane_waves_dict: dict[tuple[int, int, int],* IntensityPlaneWave*]*, *Mr=1*)

Bases: object

Manages the properties and behavior of illumination patterns, including wavevectors and spatial shifts.

angles

Array of rotation angles.

Type
np.ndarray

_spatial_shifts

List of spatial shifts.

Type
list

_Mr
Number of rotations.

Type
int

Mt
Number of spatial shifts.

Type
int

waves
Dictionary of intensity plane waves.

Type
dict

wavevectors2d
List of 2D wavevectors.

Type
list

indices2d
List of 2D indices.

Type
list

wavevectors3d
List of 3D wavevectors.

Type
list

indices3d
List of 3D indices.

Type
list

rearranged_indices
Dictionary of rearranged indices.

Type
dict

xy_fourier_peaks
Set of 2D Fourier peaks.

Type
set

phase_matrix
Dictionary of all phase the relevant phase shifts.

Type
dict

property Mr

compute_expanded_lattice2d() → set[tuple[int, int]]

Compute the expanded 2D lattice of Fourier peaks
(autoconvoluiton of Fourier transform of the illumination pattern).

Returns
Set of expanded 2D lattice peaks.

Return type
set

compute_expanded_lattice3d() → set[tuple[int, int, int]]

Compute the expanded 3D lattice of Fourier peaks
(autoconvoluiton of Fourier transform of the illumination pattern).

Returns
Set of expanded 3D lattice peaks.

Return type
set

compute_phase_matrix()

Compute the dictionary of all the relevant phase shifts
(products of spatial shifts and illumination pattern spatial frequencies).

static find_ipw_from_pw(*plane_waves*) → list[*IntensityPlaneWave*]

Static method to find intensity plane waves
(i.e. Fourier transform of the illumination pattern) from plane waves.

Parameters
plane_waves (list) – List of plane waves.

Returns
List of intensity plane waves.

Return type
list

get_all_wavevectors() → list[ndarray]

Get all wavevectors for all rotations.

Returns
List of all wavevectors.

Return type
list

get_all_wavevectors_projected()

Get all projected wavevectors for all rotations.

Returns
List of all projected wavevectors.

Return type
list

get_wavevectors(*r: int*) → tuple[list[ndarray], list[tuple[int]]]

Get the wavevectors and indices for a given rotation.

Parameters
r (int) – Rotation index.

Returns
List of wavevectors and list of indices.

Return type
tuple

get_wavevectors_projected(*r: int*) → tuple[list[ndarray], list[tuple[int]]]

Get the projected wavevectors and indices for a given rotation.

Parameters
r (int) – Rotation index.

Returns
List of projected wavevectors and list of indices.

Return type
tuple

static index_frequencies(*waves_list: list[*IntensityPlaneWave*]*, *base_vector_lengths: tuple[float, float, float]*) → dict[tuple[int, int, int], *IntensityPlaneWave*]

Static method to automatically index intensity plane waves given corresponding base vector lengths.

Parameters

- waves_list (list) – List of plane waves.
- base_vector_lengths (tuple) – Base vector lengths.

Returns
Dictionary of indexed frequencies.

Return type
dict

classmethod init_from_list(*intensity_plane_waves_list: dict[tuple[int, int, int],* IntensityPlaneWave*]*, *base_vector_lengths: tuple[float, float, float]*, *Mr=1*)

Class method to initialize Illumination from a list of intensity plane waves.

Parameters

- intensity_plane_waves_list (list) – List of intensity plane waves.

- base_vector_lengths (tuple) – Base vector lengths of the illumination Fourier space Bravais lattice.
- Mr (int) – Number of rotations.

Returns
Initialized Illumination object.

Return type
Illumination

normalize_spatial_waves()

Normalize the spatial waves on zero peak (i.e., a0 = 1).

Raises
AttributeError – If zero wavevector is not found.

set_spatial_shifts_diagonally(*number: int*, *base_vectors: tuple[float, float, float]*)

Set the spatial shifts diagonally (i.e., all the spatial shifts are assumed to be on the same lin). This is the most common use in practice. Appropriate shifts for a given illumination pattern can be computed in the module 'compute_optimal_lattices.py'

Parameters

- number (int) – Number of shifts.
- base_vectors (tuple) – Base vectors for the shifts.

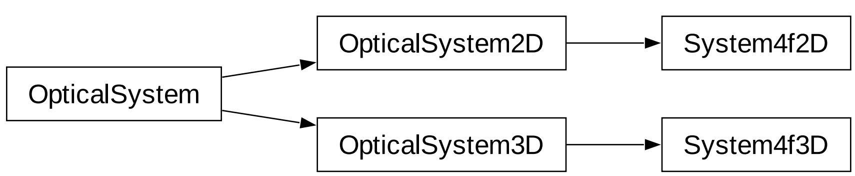
property spatial_shifts

1.1.6 OpticalSystems module

OpticalSystems.py

This module contains classes for simulating and analyzing optical systems.

Note: More reasonable interface for accessing and calculating of the PSF and OTF is expected in the future. For this reason the detailed documentation on the computations is not provided yet.



class OpticalSystems.OpticalSystem(*interpolation_method: str*)

Bases: object

Base class for optical systems, providing common functionality.

supported_interpolation_methods

List of supported interpolation methods.

Type
list

psf

Point Spread Function.

Type

np.ndarray

otf

Optical Transfer Function.

Type

np.ndarray

interpolator

Interpolator for OTF.

Type

scipy.interpolate.RegularGridInterpolator

_otf_frequencies

Frequencies for OTF.

Type

np.ndarray

_psf_coordinates

Coordinates for PSF.

Type

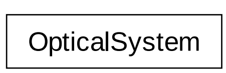
np.ndarray

_interpolation_method

Interpolation method.

Type

str



abstract compute_psf_and_otf() → tuple[ndarray[tuple[int, int, int], float64], ndarray[tuple[int, int, int], float64]]

Compute the PSF and OTF.

abstract compute_psf_and_otf_cordinates(*psf_size: tuple[int]*, *N: int*)

Compute the PSF and OTF coordinate axes.

Parameters

- psf_size (tuple) – Size of the PSF.
- N (int) – Number of points.

abstract compute_q_grid() → ndarray[tuple[int, int, int, 3], float64]

Compute the q-grid for the OTF.

Returns
Computed q-grid.

Return type
np.ndarray

abstract compute_x_grid() → ndarray[tuple[int, int, int, 3], float64]

Compute the x-grid for the PSF.

Returns
Computed x-grid.

Return type
np.ndarray

interpolate_otf(*k_shift: ndarray[3, float64]*) → ndarray[tuple[int, int, int], float64]

Interpolate the OTF with a given shift.

Parameters
k_shift (np.ndarray) – Shift vector for interpolation.

Returns
Interpolated OTF.

Return type
np.ndarray

property interpolation_method

property otf_frequencies

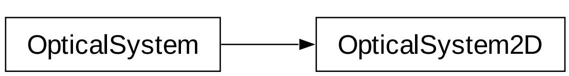
abstract property psf_coordinates

supported_interpolation_methods = ['linear', 'Fourier']

A list of currently supported interpolation methods. Other scipy interpolation methods are not directly supported due to high memory usage. Add them to the list if needed. Currently, meaningless, but changes are expected. Fourier is interpolation is used for SIM by default. Linear interpolation is available with the "interpolate_OTF" method if needed.

class OpticalSystems.OpticalSystem2D(*interpolation_method*)

Bases: OpticalSystem



compute_effective_otfs_2dSIM(*illumination:* Illumination) → dict[tuple[int, tuple[int]], float64]

Compute the effective OTFs for 2D SIM illumination (in the case of 2D SIM they are just shifted).

Parameters
illumination – Illumination object containing wave information.

Returns
Effective OTFs.

Return type
dict

compute_psf_and_otf() → tuple[float64, float64]

Compute the PSF and OTF.

compute_psf_and_otf_cordinates(*psf_size: tuple[float]*, *N: int*)

Compute the PSF and OTF coordinate axes.

Parameters

- psf_size (tuple) – Size of the PSF.
- N (int) – Number of points.

compute_q_grid() → ndarray[tuple[int, int, 2], float64]

Compute the q-grid for the OTF.

Returns
Computed q-grid.

Return type
np.ndarray

compute_x_grid() → ndarray[tuple[int, int, 2], float64]

Compute the x-grid for the PSF.

Returns
Computed x-grid.

Return type
np.ndarray

interpolate_otf(*k_shift: ndarray[3, float64]*) → ndarray[tuple[int, int, int], float64]

Interpolate the OTF with a given shift.

Parameters
k_shift (np.ndarray) – Shift vector for interpolation.

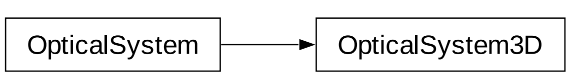
Returns
Interpolated OTF.

Return type
np.ndarray

property psf_coordinates

class OpticalSystems.OpticalSystem3D(*interpolation_method*)

Bases: OpticalSystem



compute_effective_otfs_projective_3dSIM(*illumination: Illumination*) → dict[tuple[int, tuple[int]], float64]

Compute the effective OTFs for projective 3D SIM illumination.

Parameters
illumination – Illumination object containing wave information.

Returns
Effective OTFs.

Return type
dict

compute_effective_otfs_true_3dSIM(*illumination: Illumination*) → dict[tuple[int, tuple[int]], float64]

Compute the effective OTFs for true 3D SIM (in the case of true 3D SIM, they are just shifted).

Parameters
illumination – Illumination object containing wave information.

Returns
Effective PSFs and OTFs.

Return type
tuple

compute_psf_and_otf() → tuple[float64, float64]

Compute the PSF and OTF.

compute_psf_and_otf_cordinates(*psf_size*, *N*)

Compute the PSF and OTF coordinate axes.

Parameters

- psf_size (tuple) – Size of the PSF.
- N (int) – Number of points.

compute_q_grid() → ndarray[tuple[int, int, int, int], float64]

Compute the q-grid for the OTF.

Returns
Computed q-grid.

Return type
np.ndarray

compute_x_grid() → ndarray[tuple[int, int, int, int], float64]

Compute the x-grid for the PSF.

Returns
Computed x-grid.

Return type
np.ndarray

interpolate_otf(*k_shift: ndarray[3, float64]*) → float64

Interpolate the OTF with a given shift.

Parameters
k_shift (np.ndarray) – Shift vector for interpolation.

Returns
Interpolated OTF.

Return type
np.ndarray

property psf_coordinates

class OpticalSystems.System4f2D(*alpha=0.7853981633974483*, *refractive_index=1*, *interpolation_method='linear'*)

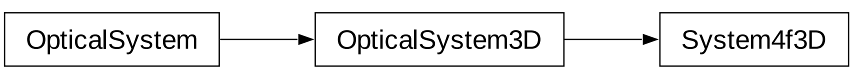
Bases: OpticalSystem2D

compute_psf_and_otf(*parameters=None*, *pupil_function=None*, *mask=None*)

Compute the PSF and OTF.

class OpticalSystems.System4f3D(*alpha=0.7853981633974483*, *refractive_index_sample=1*, *refractive_index_medium=1*, *regularization_parameter=0.01*, *interpolation_method='linear'*)

Bases: OpticalSystem3D



compute_psf_and_otf(*parameters=None*, *high_NA=False*, *apodization_function='Sine'*, *pupil_function=<function System4f3D.<lambda>>*, *mask=None*)

Compute the PSF and OTF.

1.1.7 ProcessorSIM module

ProcessorSIM.py

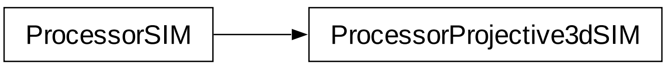
When implemented, this class will be a top-level class, responsible for SIM reconstructions.

Classes:

ProcessorSIM: Base class for SIM processors. ProcessorProjective3dSIM: Class for processing projective 3D SIM data. ProcessorTrue3dSIM: Class for processing true 3D SIM data.

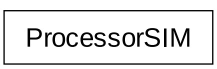
class ProcessorSIM.ProcessorProjective3dSIM(*illumination*, *optical_system*)

Bases: ProcessorSIM



class ProcessorSIM.ProcessorSIM(*illumination*, *optical_system*)

Bases: object



compute_apodization_filter_autoconvolution()

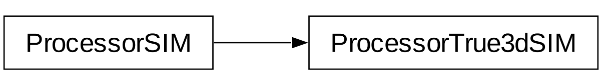
compute_apodization_filter_lukosz()

abstract static compute_effective_psfs_and_otfs(*illumination*, *optical_system*)

compute_sim_support()

class ProcessorSIM.ProcessorTrue3dSIM(*illumination*, *optical_system*)

Bases: ProcessorSIM



1.1.8 SIMulator module

SIMulator.py

This module contains the SIMulator class for simulating raw structured illumination microscopy (SIM) images and/or reconstructing the super resolution images from the raw SIM images.

This class will be probably split into two classes in the future. The detailed documentation will be provided in the further release.

class SIMulator.SIMulator(*illumination*, *optical_system*, *box_size=10*, *point_number=100*, *readout_noise_variance=0*, *additional_info=None*)

Bases: BoxSIM

generate_sim_images(*object*)

generate_sim_images2d(*object*)

generate_widefield(*sim_images*)

reconstruct_Fourier2d_finite_kernel(*sim_images*, *shifted_kernels*)

reconstruct_Fourier_space(*sim_images*)

reconstruct_real2d_finite_kernel(*sim_images*, *kernel*, *mode='same'*)

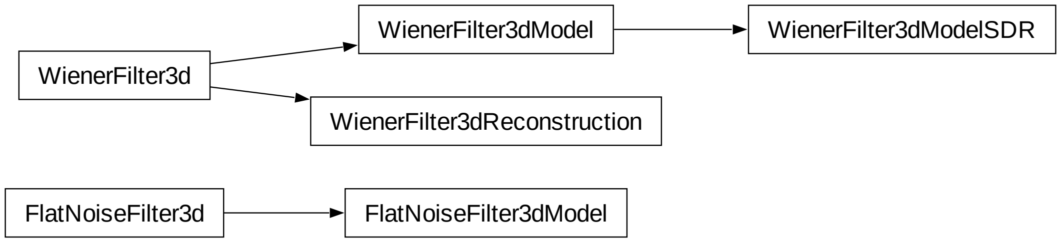
reconstruct_real_space(*sim_images*, *mode='same'*)

1.1.9 SSNRBasedFiltering module

SSNRBasedFiltering.py

This module contains classes for filtering images based on their total SSNR.

The detailed documentation for this class will be provided in the further release.



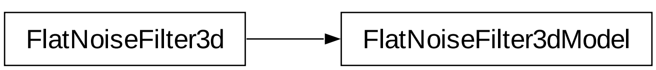
class SSNRBasedFiltering.FlatNoiseFilter3d(*ssnr_calculator*, *apodization_filter=1*)

Bases: object

filter_object(*object*, *real_space=True*)

class SSNRBasedFiltering.FlatNoiseFilter3dModel(*ssnr_calculator*, *apodization_filter=1*)

Bases: FlatNoiseFilter3d



filter_object(*model_object*, *real_space=True*)

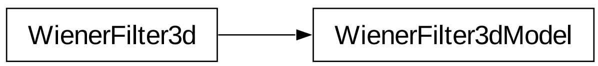
class SSNRBasedFiltering.WienerFilter3d(*ssnr_calculator*, *apodization_filter=1*)

Bases: object

filter_object(*object*, *real_space=True*)

class SSNRBasedFiltering.WienerFilter3dModel(*ssnr_calculator*, *apodization_filter=1*)

Bases: WienerFilter3d



filter_object(*model_object*, *real_space=True*)

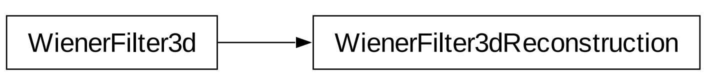
class SSNRBasedFiltering.WienerFilter3dModelSDR(*ssnr_calculator*, *apodization_filter=1*)

Bases: WienerFilter3dModel

filter_SDR_reconstruction(*object*, *reconstruction*)

class SSNRBasedFiltering.WienerFilter3dReconstruction(*ssnr_calculator*, *apodization_filter=1*)

Bases: WienerFilter3d



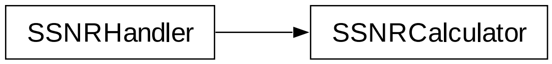
filter_object(*reconstruction*, *real_space=True*, *average='surface_levels_3d'*)

1.1.10 SSNRCalculator module

SSNRCalculator.py

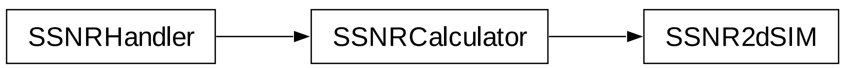
This module contains classes for calculating the (image-independent) spectral signal-to-noise ratio (SSNR) for a given system optical system and illumination.

Mathematical details will be provided in the later documentation versions and in the corresponding papers.



class SSNRCalculator.SSNR2dSIM(*illumination*, *optical_system*, *readout_noise_variance=0*)

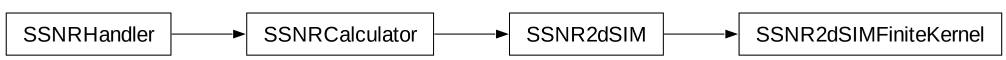
Bases: SSNRCalculator



ring_average_ssnr(*number_of_samples=None*)

class SSNRCalculator.SSNR2dSIMFiniteKernel(*illumination*, *optical_system*, *kernel*, *readout_noise_variance=0*)

Bases: SSNR2dSIM



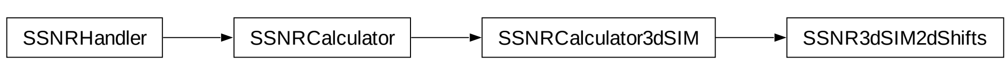
property illumination

property kernel

plot_effective_kernel_and_otf()

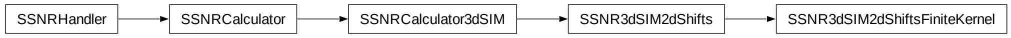
class SSNRCalculator.SSNR3dSIM2dShifts(*illumination*, *optical_system*, *readout_noise_variance=0*)

Bases: SSNRCalculator3dSIM



class SSNRCalculator.SSNR3dSIM2dShiftsFiniteKernel(*illumination*, *optical_system*, *kernel*, *readout_noise_variance=0*)

Bases: SSNR3dSIM2dShifts



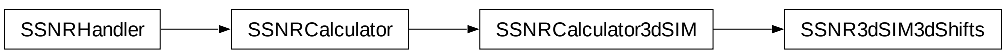
property illumination

property kernel

plot_effective_kernel_and_otf()

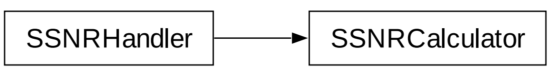
class SSNRCalculator.SSNR3dSIM3dShifts(*illumination*, *optical_system*, *readout_noise_variance=0*)

Bases: SSNRCalculator3dSIM



class SSNRCalculator.SSNRCalculator(*illumination*, *optical_system*, *readout_noise_variance=0*)

Bases: SSNRHandler



compute_analytic_ssnr_volume(*factor=10*, *volume_element=1*)

compute_analytic_total_ssnr(*factor=10*, *volume_element=1*)

compute_maximum_resolved_lateral()

compute_ssnr()

compute_ssnr_waterline_measure(*factor=10*)

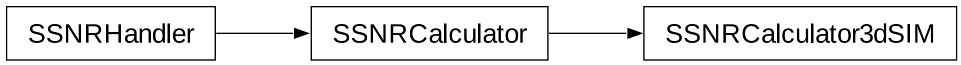
compute_total_ssnr(*factor=10*, *volume_element=1*)

property illumination

property optical_system

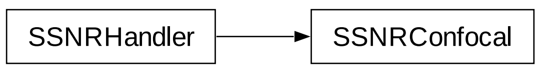
class SSNRCalculator.SSNRCalculator3dSIM(*illumination*, *optical_system*, *readout_noise_variance=0*)

Bases: SSNRCalculator



class SSNRCalculator.SSNRConfocal(*optical_system*)

Bases: SSNRHandler



compute_ssnr()

class SSNRCalculator.SSNRHandler(*optical_system*)

Bases: object

compute_radial_ssnr_entropy(*factor=100*)

abstract compute_ssnr()

compute_ssnr_volume(*factor=10*, *volume_element=1*)

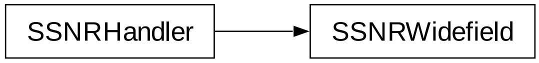
compute_true_ssnr_entropy(*factor=100*)

property optical_system

abstract ring_average_ssnr(*number_of_samples=None*)

class SSNRCalculator.SSNRWidefield(*optical_system*)

Bases: SSNRHandler



compute_ssnr()

1.1.11 ShapesGenerator module

ShapesGenerator.py

This module contains functions for generating various simulated images used in simulations.

ShapesGenerator.generate_random_lines(*image_size: tuple[int, int, int]*, *point_number: int*, *line_width: float*, *num_lines: int*, *intensity: float*) → ndarray

Generate an image with randomly oriented lines.

Parameters

- point_number – Number of points defining the size of the image grid (image will be point_number x point_number).
- image_size – Tuple of (psf_x_size, psf_y_size) defining scaling in x and y directions.
- line_width – Width of the lines.
- num_lines – Number of lines to generate.
- intensity – Total intensity of each line.

Returns

Generated image with lines.

ShapesGenerator.generate_random_spheres(*image_size: tuple[int, int, int]*, *point_number: int*, *r=0.1*, *N=10*, *I=1000*) → ndarray

Generates an array with random spheres.

Parameters

- image_size (tuple[int, int, int]) – Size of the point spread function in each dimension.
- point_number (int) – Number of points in each dimension.
- r (float, optional) – Radius of the spheres. Defaults to 0.1.
- N (int, optional) – Number of spheres to generate. Defaults to 10.
- I (int, optional) – Intensity of the spheres. Defaults to 1000.

Returns

Array with random spheres.

Return type

np.ndarray

ShapesGenerator.generate_sphere_slices(*image_size: tuple[int, int, int]*, *point_number: int*, *r=0.1*, *N=10*, *I=1000*) → ndarray

Generates a thin slice with random spheres.

Parameters

- image_size (tuple[int, int, int]) – Size of the point spread function in each dimension.
- point_number (int) – Number of points in each dimension.
- r (float, optional) – Radius of the spheres. Defaults to 0.1.
- N (int, optional) – Number of spheres to generate. Defaults to 10.
- I (int, optional) – Intensity of the spheres. Defaults to 1000.

Returns

A thin slice of random spheres.

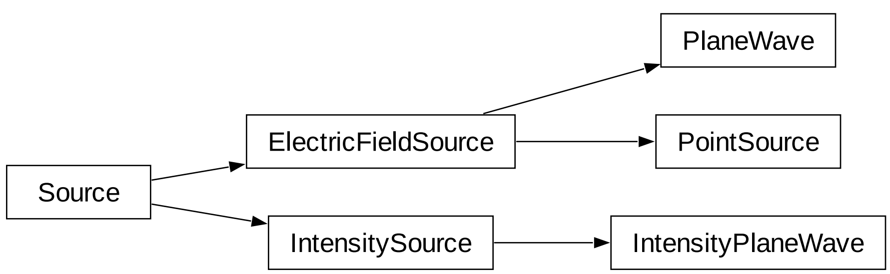
Return type

np.ndarray

1.1.12 Sources module

Sources.py

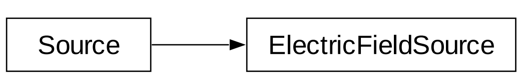
This module contains classes for different types of sources used in simulations. The sources can provide either electric fields or intensity fields.



class Sources.ElectricFieldSource

Bases: Source

Abstract base class for sources that provide an electric field.



abstract get_electric_field(*coordinates: float64*) → complex128

Gets the electric field at the given coordinates.

Parameters
coordinates (numpy.ndarray[np.float64]) – The coordinates at which to get the electric field.

Returns
The electric field at the given coordinates.

Return type
numpy.ndarray[np.complex128]

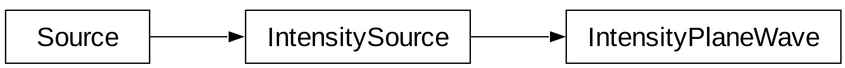
get_source_type() → str

Returns a type of the source in a human-readable form.
str: The type of the source.

class Sources.IntensityPlaneWave(*amplitude=0.0*, *phase=0.0*, *wavevector=array([0., 0., 0.])*)

Bases: IntensitySource

Intensity plane wave is a component of the Fourier transform of the energy density distribution in a given volume (e.g., standing waves)



get_intensity(*coordinates: float64*)

Gets the intensity at the given coordinates.

Parameters
coordinates (numpy.ndarray[np.float64]) – The coordinates at which to get the intensity.

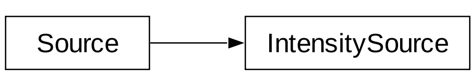
Returns
The intensity at the given coordinates.

Return type
numpy.ndarray[np.float64]

class Sources.IntensitySource

Bases: Source

Abstract base class for sources that provide intensity.



abstract get_intensity(*coordinates: float64*) → int64

Gets the intensity at the given coordinates.

Parameters
coordinates (numpy.ndarray[np.float64]) – The coordinates at which to get the intensity.

Returns
The intensity at the given coordinates.

Return type
numpy.ndarray[np.float64]

get_source_type() → str

Returns a type of the source in a human-readable form.
str: The type of the source.

class Sources.PlaneWave(*electric_field_p: complex*, *electric_field_s: complex*, *phase1: float*, *phase2: float*, *wavevector: float64*)

Bases: ElectricFieldSource

Electric field of a plane wave

get_electric_field(*coordinates*)

Gets the electric field at the given coordinates.

Parameters
coordinates (numpy.ndarray[np.float64]) – The coordinates at which to get the electric field.

Returns
The electric field at the given coordinates.

Return type
numpy.ndarray[np.complex128]

class Sources.PointSource(*coordinates: float64*, *brightness: float*)

Bases: ElectricFieldSource

Electric field of a point source

get_electric_field(*coordinates: float64*)

Gets the electric field at the given coordinates.

Parameters

coordinates (numpy.ndarray[np.float64]) – The coordinates at which to get the electric field.

Returns

The electric field at the given coordinates.

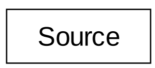
Return type

numpy.ndarray[np.complex128]

class Sources.Source

Bases: object

Abstract base class for sources of electric or intensity fields in our simulations.



abstract get_source_type() → str

Returns a type of the source in a human-readable form.

str: The type of the source.

1.1.13 VectorOperations module

VectorOperations.py

This module contains utility functions for vector operations.

Classes:

VectorOperations: Class containing static methods for various vector operations.

class VectorOperations.VectorOperations

Bases: object

VectorOperations

static rotate_vector2d(*vector2d*, *angle*)

static rotate_vector3d(*vector3d*, *rot_ax_vector*, *rot_angle*)

static rotation_matrix(*angle*)

1.1.14 Windowing module

This module provides functions to modify the image near the edges for different purposes.

Windowing.make_mask_cosine_edge2d(*shape: tuple[int, int]*, *edge: int*) → ndarray

2D Weight mask that vanishes with the cosine distance to the edge.

Parameters

- shape (tuple[int, int]) – Shape of the mask.
- edge (int) – Width of the edge.

Returns

The mask.

Return type

np.ndarray

Windowing.make_mask_cosine_edge3d(*shape: tuple[int, int, int]*, *edge: int*) → ndarray

3D Weight mask that vanishes with the cosine distance to the edges.

Parameters

- shape (tuple[int, int, int]) – Shape of the mask.
- edge (int) – Width of the edge.

Returns

The mask.

Return type

np.ndarray

1.1.15 compute_optimal_lattices module

Yet not finalized module for computing one-dimension spatial shifts, satisfying the orthogonality condition. Implemented for 2D and 3D lattices. The design is to be changed, thus no detailed documentation is provided.

compute_optimal_lattices.check_peaks2d(*matrix*, *peaks*)

compute_optimal_lattices.check_peaks3d(*matrix*, *peaks*)

compute_optimal_lattices.combine_dict(*d1*, *d2*) compute_optimal_lattices.exponent_sum2d(*matrix*, *Mx*, *My*)

compute_optimal_lattices.exponent_sum3d(*matrix*, *Mx*, *My*, *Mz*)

compute_optimal_lattices.find_pairs2d(*table2d*, *modulos*, *power1=1*)

compute_optimal_lattices.find_pairs3d(*table3d*, *modulos*, *p1=1*)

compute_optimal_lattices.find_pairs_extended(*tables*, *modulos*)

compute_optimal_lattices.generate_conditions2d(*peaks2d*)

compute_optimal_lattices.generate_conditions3d(*peaks3d*)

compute_optimal_lattices.generate_table2d(*funcs*, *bases*, *p1=1*)

compute_optimal_lattices.generate_table3d(*funcs*, *bases*, *p1=1*)

compute_optimal_lattices.generate_tables2d(*funcs*, *max_power*)

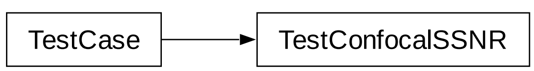
compute_optimal_lattices.get_matrix2d(*base*, *powers*)

compute_optimal_lattices.get_matrix3d(*base*, *powers*)

1.1.16 confocal_ssnr module

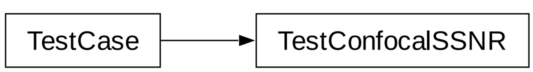
confocal_ssnr.py

This script contains test computations of the SSNR in confocal microscopy, ISM and Rescan.



class confocal_ssnr.TestConfocalSSNR(*methodName='runTest'*)

Bases: TestCase



test_SSNR2D()

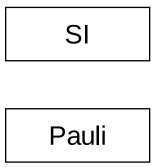
test_SSNR3D()

1.1.17 globvar module

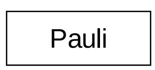
globvar.py

This module contains global variables and constants used throughout the project. It also contains physical constants and units for the case when calculations must be performed in SI units.

Classes:
Pauli: Class containing Pauli matrices. SI: Class containing various physical constants and units.



class globvar.Pauli
Bases: object



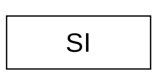
I = array([[1, 0], [0, 1]])

X = array([[0, 1], [1, 0]])

Y = array([[0.+0.j, -0.-1.j], [0.+1.j, 0.+0.j]])

Z = array([[1, 0], [0, -1]])

class globvar.SI
Bases: object



class Constants
Bases: object

Kcd = 683

NAvogadro = 6.02214076e-23

c = 299792458

dnuCs = 9192631770

e = 1.6021766340000001e-19

h = 6.62607015e-34

k = 1.380649e-23

class Energy

Bases: object

GJ = 1000000000

J = 1

MJ = 1000000

aJ = 1e-18

eV = 0.00012897410387530461

fJ = 1e-15

kJ = 1000

mJ = 0.001

mcJ = 1e-06

nJ = 1e-09

pJ = 1e-12

class Force

Bases: object

GN = 1000000000

MN = 1000000

N = 1

fN = 1e-15

kN = 1000

mN = 0.001

mcN = 1e-06

nN = 1e-09

pN = 1e-12

class Frequency

Bases: object

EHz = 1000000000000000000

GHz = 1000000000

Hz = 1

MHz = 1000000

PHz = 1000000000000000

THz = 1000000000000

kHz = 1000

class Length

Bases: object

fm = 1e-15

km = 1000

m = 1

mcm = 1e-06

mm = 0.001

nm = 1e-09

pm = 1e-12

class Time

Bases: object

ats = 1e-18

fs = 1e-15

mcs = 1e-06

ms = 0.001

ns = 1e-09

ps = 1e-12

s = 1

1.1.18 input_parser module

This module contains a class for parsing command line arguments for the initialization of GUI

class input_parser.ConfigParser

Bases: object

static read_configuration(*file*)

1.1.19 kernels module

kernels.py

This module contains functions for generating finite size real space kernels for the SSNR calculations.

Functions

sinc_kernel: Generate a 2D/3D triangular kernel, resulting in :math: *sinc^2* in Fourier space. psf_kernel2d: Generate a 2D kernel that has the shape of PSF in the Fourier domain (and hence the shape of OTF in the real space).

kernels.psf_kernel2d(*kernel_size: int*, *pixel_size: float*, *dense_kernel_size=50*) → ndarray

Generate a 2D kernel that has the shape of PSF in the Fourier domain (and hence the shape of OTF in the real space).

Parameters

- kernel_size – The size of the kernel.
- pixel_size – The pixel size in the real space.
- dense_kernel_size – The size of the dense kernel. Default is 50. This parameter is used for better interpolation of the PSF values on a small grid.

Returns

A 2D kernel.

kernels.sinc_kernel(*kernel_r_size: int*, *kernel_z_size=1*) → ndarray

Generate a 2D/3D triangular kernel, resulting in :math: *sinc^2* in Fourier space.

Parameters

- kernel_r_size – The size of the kernel in the radial direction.
- kernel_z_size – The size of the kernel in the axial direction. Default is 1.

Returns
A 2D/3D triangular kernel.

1.1.20 stattools module

stattools.py

This module contains commonly used operations on arrays, required in the context of our work.

stattools.average_mask(*array: ndarray[float64]*, *mask: ndarray[int32]*, *shape='same'*) → ndarray[float64]

Averages an array along the surface levels of the mask.

Parameters

- array (np.ndarray) – Array to average.
- mask (np.ndarray[np.int32]) – Mask indicating regions to average.
- shape (str, optional) – Shape of the output array. Defaults to 'same'.

Returns
Averaged array.

Return type
np.ndarray

stattools.average_rings2d(*array: ndarray*, *axes: tuple[ndarray] = None*, *num_angles=360*, *number_of_samples: int = None*)

Averages the 2D array radially using bilinear interpolation in polar coordinates.

Parameters

- array – 2D numpy array to average radially.
- axes – Tuple of arrays representing the grid axes (ax1, ax2).
- num_samples – Number of radial samples (r) to take.
- num_angles – Number of angular samples (theta).

Returns
Radial distances at which the interpolation is performed. averaged: Radially averaged values.

Return type
radii

stattools.average_rings3d(*array: ndarray[tuple[int, int, int], ...]*, *axes: tuple[ndarray, ndarray, ndarray] = None*) → ndarray[tuple[int, int], ...]

Averages the 3D array radially by averaging each 2D slice.

Parameters

- array (np.ndarray) – 3D array to average.
- axes (tuple, optional) – Axes for the array. Defaults to None.

Returns
Radially averaged values.

Return type
np.ndarray

stattools.downsample_circular_function_vectorized(*dense_function*, *small_size*)

Downsample a circularly symmetric function from a large grid to a smaller grid using a vectorized approach.

Parameters

- dense_function – 2D NumPy array representing the function values on the large grid (e.g., 51 x 51).
- small_size – Tuple (m, n) representing the size of the small grid (e.g., (5, 5)).

Returns
2D NumPy array representing the downsampled function on the smaller grid.

Return type
small_grid

stattools.estimate_localized_peaks(*array*, *axes*)

Estimates localized peaks in a 3D array. Current implementation is inefficient and will be replaced.

Parameters

- array (np.ndarray) – 3D array to analyze.
- axes (tuple) – Axes for the array.

Returns
Localized peaks and their amplitudes.

Return type
tuple

stattools.expand_ring_averages2d(*averaged: ndarray[int, ...]*, *axes: tuple[ndarray, ndarray] = None*) → ndarray[tuple[int, int], ...]

Expands the radially averaged 2D array back to its original shape.

Parameters

- averaged (np.ndarray) – Radially averaged values.
- axes (tuple, optional) – Axes for the array. Defaults to None.

Returns
Expanded array.

Return type
np.ndarray

stattools.expand_ring_averages3d(*averaged: ndarray[tuple[int, int], ...]*, *axes: tuple[ndarray, ndarray, ndarray] = None*) → ndarray[tuple[int, int, int], ...]

Expands the radially averaged 3D array back to its original shape.

Parameters

- averaged (np.ndarray) – Radially averaged values.
- axes (tuple, optional) – Axes for the array. Defaults to None.

Returns
Expanded array.

Return type
np.ndarray

stattools.find_decreasing_radial_surface_levels(*array*, *axes=None*)

Not implemented yet

stattools.find_decreasing_surface_levels2d(*array: ndarray[tuple[int, int], float64]*, *axes=None*, *direction=None*) → ndarray[tuple[int, int], int32]

Assuming function is monotonically decaying around some point, finds surface levels of this function. No interpolation is used.

Parameters

- array (np.ndarray) – 2D array to analyze.
- axes (tuple, optional) – Axes for the array. Defaults to None.
- direction (int, optional) – Direction to analyze. Defaults to None.

Returns
Mask indicating the surface levels.

Return type
np.ndarray

stattools.find_decreasing_surface_levels3d(*array: ndarray[tuple[int, int, int], float64]*, *axes=None*, *direction=None*) → ndarray[tuple[int, int, int], int32]

Assuming function is monotonically decaying around some point, finds surface levels of this function. No interpolation is used.

Parameters

- array (np.ndarray) – 3D array to analyze.
- axes (tuple, optional) – Axes for the array. Defaults to None.
- direction (int, optional) – Direction to analyze. Defaults to None.

Returns
Mask indicating the surface levels.

Return type
np.ndarray

stattools.gaussian_maxima_fitting(*array*, *axes*, *maxima_indices*, *size=5*)

Fits Gaussian functions to the maxima in a 3D array.

Parameters

- array (np.ndarray) – 3D array to analyze.
- axes (tuple) – Axes for the array.
- maxima_indices (list) – Indices of the maxima.
- size (int, optional) – Size of the fitting window. Defaults to 5.

Returns
Fitted maxima and their standard deviations.

Return type
tuple

stattools.reverse_interpolation_nearest(*x_axis*, *y_axis*, *points*, *values*)

Interpolate values from known points to a grid, affecting only the nearest grid cells.

Parameters

- x_axis – 1D array representing the x-coordinates of the grid.
- y_axis – 1D array representing the y-coordinates of the grid.
- points – Array of known points' coordinates, shape (N, 2).
- values – Array of known values at the points, shape (N,).

Returns
2D array of interpolated values on the grid.

Return type
interpolated_grid

1.1.21 web_interface module

Zeroth iteration on AI generated web interface.

web_interface.index()

web_interface.plot()

1.1.22 wrappers module

wrappers.py

This module contains wrapper functions for Fourier transforms to make shifts automatically and make it possible to switch between their implementations.

Functions:
wrapped_fftn: Wrapper for the FFTN function. wrapped_ifftn: Wrapper for the IFFTN function.

wrappers.wrapped_fft(*arrays*, **args*, ***kwargs*)

wrappers.wrapped_fftn(*arrays*, **args*, ***kwargs*)

wrappers.wrapped_ifft(*arrays*, **args*, ***kwargs*)

wrappers.wrapped_ifftn(*arrays*, **args*, ***kwargs*)

wrappers.wrapper_ft(*ft*)

Wrapper for the Fourier transform functions to make shifts automatically. Currently based on numpy fft implementation.

CHAPTER

TWO

API REFERENCE

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

G

N

O

P

R

S

T

V

W

X

Y

Z