NIRFASTerFF

Release 1.1.1

Jiaming Cao, MILAB@UoB

Dec 13, 2024

CONTENTS

NIRFAST was originally developed in 2001 as a MATLAB-based package used to model near-infrared light propagation through tissue and reconstruct images of optical biomarkers. At its core, NIRFAST is tool that does finite element modeling of the medium in which the light progagates, and calculates the fluence field by solving the diffusion equation.

A fully overhauled version, titled NIRFASTer, was published in 2018, thanks to the work of Dr. Stanisław Wojtkiewicz. In the new version, GPU support was added to the key algorithms, giving the software a dramatic boost in performance. The CPU versions of the algorithms were re-implemented in C++ with multithreading enabled, and the performance was improved considerably.

In this version, now titled NIRFASTerFF (Fast and Furious), the entire toolbox is re-written with Python as its interfacing language, while fully inter-operatable with the original Matlab version. The algorithms, running on both GPU and CPU, are yet again improved for even better performance.

This manual is a detailed documentation of all APIs in the package. Please also refer to the demos to see how the package is used in difference applications.

CHAPTER

ONE

SUMMARY OF THE FUNCTIONALITIES

Mesh types supported: standard, fluorecence, and DCS

FEM solver calculates: CW fluence, FD fluence, TPSF, direct TR moments for standard and fluorecence mesh, and G1/g1 curve for DCS mesh

Analytical solution in semi-infinite medium for: CW/FD fluence, TPSF, and DCS G1/g1 curves

Jacobian matrices: CW for standard, fluorescence, and DCS mesh, and FD for standard and fluorescence mesh

CHAPTER TWO

LINK TO THE MATLAB VERSION

The original Matlab-based NIRFAST and NIRFASTer are still avaibable for download, but we will gradually drop our support for them.

https://github.com/nirfast-admin/NIRFAST

https://github.com/nirfaster/NIRFASTer

CHAPTER

THREE

REFERENCES

If you use our package, please cite,

H. Dehghani, M.E. Eames, P.K. Yalavarthy, S.C. Davis, S. Srinivasan, C.M. Carpenter, B.W. Pogue, and K.D. Paulsen, “Near infrared optical tomography using NIRFAST: Algorithm for numerical model and image reconstruction,” Communications in Numerical Methods in Engineering, vol. 25, 711-732 (2009) doi:10.1002/cnm.1162

CHAPTER

FOUR

API DOCUMENTATION

4.1 nirfasterff

Modules

base	Core classes used in the pacakge
forward	Funtions for forward data calculation
inverse	Calculation of the Jacobian matrices and a basic Tikhonov regularization function
io	Some functions for reading/writing certain data types.
lib	Low-level functions implemented in C/C++, on both GPU and CPU
math	Some low-level functions used by the forward solvers.
meshing	Functions used for mesh generation and quality check
utils	Utility functions and auxiliary classes frequently used in the package.
visualize	Functions for basic data visualization

4.1.1 nirfasterff.base

Core classes used in the pacakge

Modules

data	Defining some data classes, which are the return types of the fem data calculation functions
dcs_mesh	Define the DCS mesh class.
fluor_mesh	Define the fluorescence mesh class
optodes	Define the optode class, an instance of which can be either a source or a detector
stnd_mesh	Define the standard mesh class

nirfasterff.base.data

Defining some data classes, which are the return types of the fem data calculation functions

Classes

DCSdata()	Class holding DCS data.
FDdata()	Class holding FD/CW data.
FLdata()	Class holding FD/CW fluorescence data.
TPSFdata()	Class holding time-resolved TPSF data.
TRMomentsdata()	Class holding time-resolved moments data calculated using Mellin transform.
flTPSFdata()	Class holding fluorescence time-resolved TPSF data.
flTRMomentsdata()	Class holding fluorescence TR moments data calculated using Mellin transform.
meshvol()	Small class holding the information needed for converting between mesh and volumetric space.

nirfasterff.base.data.DCSdata

class `nirfasterff.base.data.`**`DCSdata`**

Bases: `object`

Class holding DCS data.

`phi`

steady-state fluence from each source. If mesh contains non-tempty field vol, this will be represented on the grid.

Last dimension has the size of the number of sources

This is the same as nirfasterff.base.FDdata.phi, when modulation frequency is zero

Type
double NumPy array

`link`

Defining all the channels (i.e. source-detector pairs). Copied from mesh.link

Type
int32 NumPy array

`amplitude`

Steady-state amplitude of each channel. Size (NChannel,)

This is the same as nirfasterff.base.FDdata.amplitude, when modulation frequency is zero

Type
double NumPy vector

`tau_DCS`

time vector in seconds

Type
double NumPy vector

`phi_DCS`

G1 in medium from each source at each time step . If mesh contains non-tempty field vol, this will be represented on the grid

shape[-1] equals length of tau_DCS, and shape[-2] equals number of sources

Type
double NumPy array

`G1_DCS`

G1 curve as is calculated from the correlation diffusion equation. Size: (NChannel, NTime)

Type
double NumPy array

`g1_DCS`

g1 curve, i.e. G1 normalized by amplitudes. Size: (NChannel, NTime)

Type
double NumPy array

`vol`

Information needed to convert between volumetric and mesh space. Copied from mesh.vol

Type
nirfaseterff.base.meshvol

`__init__()`

Methods

`__init__`()	
`isvol`()	Checks if data is in volumetric space.
`togrid`(mesh)	Convert data to volumetric space as is defined in mesh.vol.
`tomesh`(mesh)	Convert data back to mesh space using information defined in mesh.vol.

`isvol()`

Checks if data is in volumetric space.

Returns
True if data is in volumetric space, False if not.

Return type
bool

`togrid`(*mesh*)

Convert data to volumetric space as is defined in mesh.vol. If it is empty, the function does nothing.

If data is already in volumetric space, function casts data to the new volumetric space

CAUTION: This OVERRIDES the fields phi and phi_DCS

Parameters
`mesh` (*`nirfasterff.base.dcsmesh`*) – mesh whose .vol attribute is used to do the conversion.

Return type
None.

tomesh(*mesh*)

Convert data back to mesh space using information defined in mesh.vol. If data.vol is empty, the function does nothing.

CAUTION: This OVERRIDES fields phi and phi_DCS

Parameters
: **mesh** (*nirfasterff.base.dcsmesh*) – mesh whose .vol attribute is used to do the conversion.

Return type
: None.

nirfasterff.base.data.FDdata

class nirfasterff.base.data.**FDdata**

Bases: `object`

Class holding FD/CW data.

phi

Fluence from each source. If mesh contains non-tempty field vol, this will be represented on the grid. Last dimension has the size of the number of sources

Type
: double Numpy array

complex

Complex amplitude of each channel. Same as amplitude in case of CW data

Type
: double or complex double Numpy vector

link

Defining all the channels (i.e. source-detector pairs). Copied from mesh.link

Type
: int32 NumPy array

amplitude

Absolute amplitude of each channel. I.e. amplitude=abs(complex)

Type
: double Numpy vector

phase

phase data of each channel. All zero in case of CW data

Type
: double Numpy vector

vol

Information needed to convert between volumetric and mesh space. Copied from mesh.vol

Type
: nirfaseterff.base.meshvol

__init__()

Methods

__init__()	
isvol()	Checks if data is in volumetric space.
togrid(mesh)	Convert data to volumetric space as is defined in mesh.vol.
tomesh(mesh)	Convert data back to mesh space using information defined in mesh.vol.

isvol()

Checks if data is in volumetric space.

Returns
True if data is in volumetric space, False if not.

Return type
bool

togrid(*mesh*)

Convert data to volumetric space as is defined in mesh.vol. If it is empty, the function does nothing.

If data is already in volumetric space, function casts data to the new volumetric space

CAUTION: This OVERRIDES the field phi

Parameters
mesh (*nirfasterff.base.stndmesh*) – mesh whose .vol attribute is used to do the conversion.

Return type
None.

tomesh(*mesh*)

Convert data back to mesh space using information defined in mesh.vol. If data.vol is empty, the function does nothing.

CAUTION: This OVERRIDES the field phi

Parameters
mesh (*nirfasterff.base.stndmesh*) – mesh whose .vol attribute is used to do the conversion.

Return type
None.

nirfasterff.base.data.FLdata

class nirfasterff.base.data.**FLdata**

Bases: `object`

Class holding FD/CW fluorescence data.

phix

intrinsic fluence from each source at excitation wavelength. If mesh contains non-tempty field vol, this will be represented on the grid. Last dimension has the size of the number of sources

Type
double Numpy array

`phimm`

intrinsic from each source at emission wavelength.

Type
double Numpy array

`phifl`

fluorescence emission fluence

Type
double Numpy array

`complexx`

Complex amplitude of each channel, intrinsic excitation. Same as amplitude in case of CW data

Type
double or complex double Numpy vector

`complexmm`

Complex amplitude of each channel, intrinsic emission. Same as amplitude in case of CW data

Type
double or complex double Numpy vector

`complexfl`

Complex amplitude of each channel, fluorescence emission. Same as amplitude in case of CW data

Type
double or complex double Numpy vector

`link`

Defining all the channels (i.e. source-detector pairs). Copied from mesh.link

Type
int32 NumPy array

`amplitudex`

Absolute amplitude of each channel, intrinsic excitation. I.e. amplitudex=abs(complexx)

Type
double Numpy vector

`amplitudemm`

Absolute amplitude of each channel, intrinsic emission. I.e. amplitudemm=abs(complexmm)

Type
double Numpy vector

`amplitudefl`

Absolute amplitude of each channel, fluorescence emission. I.e. amplitudefl=abs(complexfl)

Type
double Numpy vector

`phasex`

phase data of each channel, intrinsic excitation. All zero in case of CW data

Type
double Numpy vector

phasemm

phase data of each channel, intrinsic emission. All zero in case of CW data

Type
double Numpy vector

phasefl

phase data of each channel, fluorescence emission. All zero in case of CW data

Type
double Numpy vector

vol

Information needed to convert between volumetric and mesh space. Copied from mesh.vol

Type
nirfaseterff.base.meshvol

__init__()

Methods

`__init__`()	
`isvol`()	Checks if data is in volumetric space.
`togrid`(mesh)	Convert data to volumetric space as is defined in mesh.vol.
`tomesh`(mesh)	Convert data back to mesh space using information defined in mesh.vol.

isvol()

Checks if data is in volumetric space.

Returns
True if data is in volumetric space, False if not.

Return type
bool

togrid(*mesh*)

Convert data to volumetric space as is defined in mesh.vol. If it is empty, the function does nothing.

If data is already in volumetric space, function casts data to the new volumetric space

CAUTION: This OVERRIDES the fields phix, phimm, and phifl, if they are defined

Parameters
mesh (*nirfasterff.base.fluormesh*) – mesh whose .vol attribute is used to do the conversion.

Return type
None.

tomesh(*mesh*)

Convert data back to mesh space using information defined in mesh.vol. If data.vol is empty, the function does nothing.

CAUTION: This OVERRIDES fields phix, phimm, and phifl, if they are defined

Parameters
: `mesh` (*nirfasterff.base.fluormesh*) – mesh whose .vol attribute is used to do the conversion.

Return type
: None.

nirfasterff.base.data.TPSFdata

`class nirfasterff.base.data.TPSFdata`

Bases: `object`

Class holding time-resolved TPSF data.

`phi`

TPSF from each source at each spatial location. If mesh contains non-tempty field vol, this will be represented on the grid

Shape: NNodes x num_sources x time_steps

OR: len(xgrid) x len(ygrid) x len(zgrid) x num_sources x time_steps

None by default, and only contains data if 'field' option is set to True when calculating forward data.

Type
: double NumPy array or None

`time`

time vector, in seconds

Type
: double NumPy vector

`tpsf`

TPSF measured at each channel. Size: (NChannels, time_steps)

Type
: double NumPy array

`link`

Defining all the channels (i.e. source-detector pairs). Copied from mesh.link

Type
: int32 NumPy array

`vol`

Information needed to convert between volumetric and mesh space. Copied from mesh.vol

Type
: nirfaseterff.base.meshvol

`__init__`()

Methods

__init__()	
isvol()	Checks if data is in volumetric space.
togrid(mesh)	Convert data to volumetric space as is defined in mesh.vol.
tomesh(mesh)	Convert data back to mesh space using information defined in mesh.vol.

isvol()

Checks if data is in volumetric space.

Returns
True if data is in volumetric space, False if not.

Return type
bool

togrid(*mesh*)

Convert data to volumetric space as is defined in mesh.vol. If it is empty or data.phi==None, the function does nothing.

If data is already in volumetric space, function casts data to the new volumetric space

CAUTION: This OVERRIDES the field phi, if it is defined

Parameters
mesh (*nirfasterff.base.stndmesh*) – mesh whose .vol attribute is used to do the conversion.

Return type
None.

tomesh(*mesh*)

Convert data back to mesh space using information defined in mesh.vol. If data.vol is empty or data.phi==None, the function does nothing.

CAUTION: This OVERRIDES field phi, if it is defined

Parameters
mesh (*nirfasterff.base.fluormesh*) – mesh whose .vol attribute is used to do the conversion.

Return type
None.

nirfasterff.base.data.TRMomentsdata

class nirfasterff.base.data.**TRMomentsdata**

Bases: `object`

Class holding time-resolved moments data calculated using Mellin transform.

phi

moments from each source at each spatial location. If mesh contains non-tempty field vol, this will be represented on the grid

Shape: NNodes x num_sources x (max_moment_order + 1)

OR: len(xgrid) x len(ygrid) x len(zgrid) x num_sources x (max_moment_order + 1)

None by default, and only contains data if 'field' option is set to True when calculating forward data.

Type
double Numpy array or None

`moments`

moments for each channel. i-th column contains i-th moment as measured at each channel. Size: (NChannels, max_moment_order + 1)

Type
double Numpy vector

`link`

Defining all the channels (i.e. source-detector pairs). Copied from mesh.link

Type
int32 NumPy array

`vol`

Information needed to convert between volumetric and mesh space. Copied from mesh.vol

Type
nirfaseterff.base.meshvol

`__init__()`

Methods

__init__()	
isvol()	Checks if data is in volumetric space.
togrid(mesh)	Convert data to volumetric space as is defined in mesh.vol.
tomesh(mesh)	Convert data back to mesh space using information defined in mesh.vol.

`isvol()`

Checks if data is in volumetric space.

Returns
True if data is in volumetric space, False if not.

Return type
bool

`togrid`(*mesh*)

Convert data to volumetric space as is defined in mesh.vol. If it is empty or data.phi==None, the function does nothing.

If data is already in volumetric space, function casts data to the new volumetric space

CAUTION: This OVERRIDES the field phi, if it is defined

Parameters
mesh (*nirfasterff.base.stndmesh*) – mesh whose .vol attribute is used to do the conversion.

Return type
None.

`tomesh`(*mesh*)

Convert data back to mesh space using information defined in mesh.vol. If data.vol is empty or data.phi==None, the function does nothing.

CAUTION: This OVERRIDES field phi, if it is defined

Parameters
mesh (*nirfasterff.base.fluormesh*) – mesh whose .vol attribute is used to do the conversion.

Return type
None.

nirfasterff.base.data.flTPSFdata

class nirfasterff.base.data.**flTPSFdata**

Bases: `object`

Class holding fluorescence time-resolved TPSF data.

phix

TPSF at exciation wavelength from each source at each spatial location. If mesh contains non-tempty field vol, this will be represented on the grid

Shape: NNodes x num_sources x time_steps

OR: len(xgrid) x len(ygrid) x len(zgrid) x num_sources x time_steps

None by default, and only contains data if 'field' option is set to True when calculating forward data.

Type
double NumPy array

phifl

similar to phix, but for fluorescence emission

Type
double NumPy array

time

time vector, in seconds

Type
double NumPy vector

tpsfx

TPSF measured at each channel, excitation. Size: (NChannels, time_steps)

Type
double NumPy array

link

Defining all the channels (i.e. source-detector pairs). Copied from mesh.link

Type
int32 NumPy array

`vol`

Information needed to convert between volumetric and mesh space. Copied from mesh.vol

Type
nirfaseterff.base.meshvol

`__init__()`

Methods

__init__()	
isvol()	Checks if data is in volumetric space.
togrid(mesh)	Convert data to volumetric space as is defined in mesh.vol.
tomesh(mesh)	Convert data back to mesh space using information defined in mesh.vol.

`isvol()`

Checks if data is in volumetric space.

Returns
True if data is in volumetric space, False if not.

Return type
bool

`togrid`(*mesh*)

Convert data to volumetric space as is defined in mesh.vol. If it is empty or data.phix==None, the function does nothing.

If data is already in volumetric space, function casts data to the new volumetric space

CAUTION: This OVERRIDES the fields phix and phifl, if they are defined

Parameters
mesh (*nirfasterff.base.fluormesh*) – mesh whose .vol attribute is used to do the conversion.

Return type
None.

`tomesh`(*mesh*)

Convert data back to mesh space using information defined in mesh.vol. If data.vol is empty or data.phix==None, the function does nothing.

CAUTION: This OVERRIDES fields phix and phifl, if they are defined

Parameters
mesh (*nirfasterff.base.fluormesh*) – mesh whose .vol attribute is used to do the conversion.

Return type
None.

nirfasterff.base.data.flTRMomentsdata

`class nirfasterff.base.data.`**`flTRMomentsdata`**

Bases: `object`

Class holding fluorescence TR moments data calculated using Mellin transform.

`phix`

moments at excitation wavelength from each source at each spatial location. If mesh contains non-tempty field vol, this will be represented on the grid

Shape: NNodes x num_sources x (max_moment_order + 1)

OR: len(xgrid) x len(ygrid) x len(zgrid) x num_sources x (max_moment_order + 1)

None by default, and only contains data if 'field' option is set to True when calculating forward data.

Type
double NumPy array

`phifl`

similar to phix, but for fluorescence emission

Type
double NumPy array

`momentsx`

moments for each channel, exciation. i-th column contains i-th moment. Size: (NChannels, max_moment_order + 1)

Type
double NumPy array

`momentsfl`

moments for each channel, fluorescence emission. i-th column contains i-th moment. Size: (NChannels, max_moment_order + 1)

Type
double NumPy array

`link`

Defining all the channels (i.e. source-detector pairs). Copied from mesh.link

Type
int32 NumPy array

`vol`

Information needed to convert between volumetric and mesh space. Copied from mesh.vol

Type
nirfaseterff.base.meshvol

`__init__``()`

Methods

__init__()	
isvol()	Checks if data is in volumetric space.
togrid(mesh)	Convert data to volumetric space as is defined in mesh.vol.
tomesh(mesh)	Convert data back to mesh space using information defined in mesh.vol.

isvol()

Checks if data is in volumetric space.

Returns
True if data is in volumetric space, False if not.

Return type
bool

togrid(*mesh*)

Convert data to volumetric space as is defined in mesh.vol. If it is empty or data.phix==None, the function does nothing.

If data is already in volumetric space, function casts data to the new volumetric space

CAUTION: This OVERRIDES the fields phix and phifl, if they are defined

Parameters
mesh (*nirfasterff.base.fluormesh*) – mesh whose .vol attribute is used to do the conversion.

Return type
None.

tomesh(*mesh*)

Convert data back to mesh space using information defined in mesh.vol. If data.vol is empty or data.phix==None, the function does nothing.

CAUTION: This OVERRIDES fields phix and phifl, if they are defined

Parameters
mesh (*nirfasterff.base.fluormesh*) – mesh whose .vol attribute is used to do the conversion.

Return type
None.

nirfasterff.base.data.meshvol

class nirfasterff.base.data.**meshvol**

Bases: `object`

Small class holding the information needed for converting between mesh and volumetric space. Values calculated by nirfasterff.base.*mesh.gen_intmat

Note that the volumetric space, defined by xgrid, ygrid, and zgrid (empty for 2D mesh), must be uniform `xgrid`

x grid of the volumetric space. In mm

Type
double Numpy array

`ygrid`

y grid of the volumetric space. In mm

Type
double Numpy array

`zgrid`

z grid of the volumetric space. In mm. Empty for 2D meshes

Type
double Numpy array

`mesh2grid`

matrix converting a vector in mesh space to volumetric space, done by mesh2grid.dot(data)

The result is vectorized in 'F' (Matlab) order

Size: (len(xgrid)*len(ygrid)*len(zgrid), NNodes)

Type
double CSR sparse matrix

`gridinmesh`

indices (one-based) of data points in the volumetric space that are within the mesh space, vectorized in 'F' order.

Type
int32 NumPy array

`res`

resolution in x, y, z (if 3D) direction, in mm. Size (2,) or (3,)

Type
double NumPy array

`grid2mesh`

matrix converting volumetric data, vectorized in 'F' order, to mesh space. Done by grid2mesh.dot(data)

Size (Nnodes, len(xgrid)*len(ygrid)*len(ygrid))

Type
double CSR sparse matrix

`meshingrid`

indices (one-based) of data points in the mesh space that are within the volumetric space

Type
int32 NumPy array

`__init__()`

Methods

__init__()	

nirfasterff.base.dcs_mesh

Define the DCS mesh class. Assuming all motions are Brownian.

Classes

dcsmesh()	Main class for standard mesh.

nirfasterff.base.dcs_mesh.dcsmesh

`class nirfasterff.base.dcs_mesh.`**`dcsmesh`**

Bases: `object`

Main class for standard mesh. The methods should cover most of the commonly-used functionalities

name

name of the mesh. Default: 'EmptyMesh'

Type
str

nodes

locations of nodes in the mesh. Unit: mm. Size (NNodes, dim)

Type
double NumPy array

bndvtx

indicator of whether a node is at boundary (1) or internal (0). Size (NNodes,)

Type
double NumPy array

type

type of the mesh. It is always 'dcs'.

Type
str

mua

absorption coefficient (mm^-1) at each node. Size (NNodes,)

Type
double NumPy array

kappa

diffusion coefficient (mm) at each node. Size (NNodes,). Defined as 1/(3*(mua + mus))

Type
double NumPy array

`ri`

refractive index at each node. Size (NNodes,)

Type
double NumPy array

`mus`

reduced scattering coefficient (mm^-1) at each node. Size (NNodes,)

Type
(double NumPy array

`wv_DCS`

wavelength used (nm)

Type
double

`alpha`

fraction of photon scattering events that occur from moving particles in the medium (a.u.). Size (NNodes, NFlow)

Type
double NumPy array

`Db`

effective diffusion coefficient in Brownian motion (mm^2/s). Size (NNodes, NFlow)

Type
double NumPy array

`aDb`

Defined as np.sum(a*Db,axis=1). This lumped parameter (mm^2/s) is what is actually used in data genreation. Size (NNodes,)

Type
double NumPy array

`elements`

triangulation (tetrahedrons or triangles) of the mesh, Size (NElements, dim+1)

Row i contains the indices of the nodes that form tetrahedron/triangle i

One-based indexing for direct interoperatability with the Matlab version

Type
double NumPy array

`region`

region labeling of each node. Starting from 1. Size (NNodes,)

Type
double NumPy array

`source`

information about the sources

Type
nirfasterff.base.optode

`meas`

information about the the detectors

Type
nirfasterff.base.optode

`link`

list of source-detector pairs, i.e. channels. Size (NChannels,3)

First column: source; Second column: detector; Third column: active (1) or not (0)

Type
int32 NumPy array

`c`

light speed (mm/sec) at each node. Size (NNodes,). Defined as c0/ri, where c0 is the light speed in vacuum

Type
double NumPy array

`ksi`

photon fluence rate scale factor on the mesh-outside_mesh boundary as derived from Fresenel's law. Size (NNodes,)

Type
double NumPy array

`element_area`

volume/area (mm^3 or mm^2) of each element. Size (NElements,)

Type
double NumPy array

`support`

total volume/area of all the elements each node belongs to. Size (NNodes,)

Type
double NumPy array

`vol`

object holding information for converting between mesh and volumetric space.

Type
nirfasterff.base.meshvol

`__init__()`

Methods

`__init__`()	
`change_prop`(idx, prop)	Change optical properties (mua, musp, ri, alpha and Db) at nodes specified in idx, and automatically change fields kappa, c, and ksi as well
`femdata`(tvec[, solver, opt])	Calculates steady-state fluences and G1/g1 curves for each source using a FEM solver, and then the boudary measurables for each channel
`from_copy`(mesh)	Deep copy all fields from another mesh.
`from_file`(file)	Read from classic NIRFAST mesh (ASCII) format, not checking the correctness of the loaded integration functions.
`from_mat`(matfile[, varname])	Read from Matlab .mat file that contains a NIRFASTer mesh struct.
`from_solid`(ele, nodes[, prop, src, det, link])	Construct a NIRFASTer mesh from a 3D solid mesh generated by a mesher.
`from_volume`(vol[, param, prop, src, det, link])	Construct mesh from a segmented 3D volume using the built-in CGAL mesher.
`gen_intmat`(xgrid, ygrid[, zgrid])	Calculate the information needed to convert data between mesh and volumetric space, specified by x, y, z (if 3D) grids.
`isvol`()	Check if convertion matrices between mesh and volumetric spaces are calculated
`jacobian`(tvec[, normalize, solver, opt])	Calculates the Jacobian matrix
`save_nirfast`(filename)	Save mesh in the classic NIRFASTer ASCII format, which is directly compatible with the Matlab version
`set_prop`(prop)	Set optical properties of the whole mesh, using information provided in prop.
`touch_optodes`()	Moves all optodes (if non fixed) and recalculate the integration functions (i.e. barycentric coordinates).

change_prop(*idx*, *prop*)

Change optical properties (mua, musp, ri, alpha and Db) at nodes specified in idx, and automatically change fields kappa, c, and ksi as well

Parameters

- **idx** (*list or NumPy array or -1*) – zero-based indices of nodes to change. If *idx==-1*, function changes all the nodes
- **prop** (*list or NumPy array of length 6*) – new optical properties to be assigned to the specified nodes. [region mua(mm-1) musp(mm-1) ri alpha Db].

Return type

None.

femdata(*tvec*, *solver=utils.get_solver()*, *opt=utils.SolverOptions()*)

Calculates steady-state fluences and G1/g1 curves for each source using a FEM solver, and then the boudary measurables for each channel

Assumes Brownian motion, that is, $\langle \Delta r^2 \rangle = 6 * \alpha Db * \tau$

If *mesh.vol* is set, fluence and G1 data will be represented in volumetric space

See `femdata_DCS()` and `femdata_stnd_CW()` for details

Parameters

- **tvec** (*double NumPy array*) – time vector (i.e. :math:` au`) for the G1 curve, in seconds. It is usually a good idea to use log scale
- **solver** (*str, optional*) – Choose between 'CPU' or 'GPU' solver (case insensitive). Automatically determined (GPU prioritized) if not specified
- **opt** (`nirfasterff.utils.SolverOptions`, *optional*) – Solver options. Uses default parameters if not specified, and they should suffice in most cases.

  See *SolverOptions()* for details

Returns

- **data** (*nirfasterff.base.DCSdata*) – contains fluence, G1 curve, and g1 curve calculated at each spatial location, and also the boundary data.

  See *DCSdata()* for details.
- **info** (*nirfasterff.utils.ConvergenceInfo*) – convergence information of the solver when calculating the fluence field.

  See *ConvergenceInfo()* for details

from_copy(*mesh*)

Deep copy all fields from another mesh.

Parameters
mesh (*nirfasterff.base.stndmesh*) – the mesh to copy from.

Return type
None.

from_file(*file*)

Read from classic NIRFAST mesh (ASCII) format, not checking the correctness of the loaded integration functions.

All fields after loading should be directly compatible with Matlab version.

Parameters
file (*str*) – name of the mesh. Any extension will be ignored.

Return type
None.

Examples

```
>>> mesh = nirfasterff.base.dcsmesh()
>>> mesh.from_file('meshname')
```

from_mat(*matfile*, *varname=None*)

Read from Matlab .mat file that contains a NIRFASTer mesh struct. All fields copied as is without error checking.

Parameters

- **matfile** (*str*) – name of the .mat file to load. Use of extension is optional.
- **varname** (*str, optional*) – if your .mat file contains multiple variables, use this argument to specify which one to load. The default is None.

When *varname==None*, *matfile* should contain exatly one structure, which is a NIRFASTer mesh, or the function will do nothing

Return type
None.

from_solid(*ele*, *nodes*, *prop=None*, *src=None*, *det=None*, *link=None*)

Construct a NIRFASTer mesh from a 3D solid mesh generated by a mesher. Similar to the solidmesh2nirfast function in Matlab version.

Can also set the optical properties and optodes if supplied

Parameters

- **ele** (*int/double NumPy array*) – element list in one-based indexing. If four columns, all nodes will be labeled as region 1

  If five columns, the last column will be used for region labeling.

- **nodes** (*double NumPy array*) – node locations in the mesh. Unit: mm. Size (NNodes,3).

- **prop** (*double NumPy array, optional*) – If not *None*, calls *dcsmesh.set_prop()* and sets the optical properties in the mesh. The default is None.

  See *set_prop()* for details.

- **src** (*nirfasterff.base.optode, optional*) – If not *None*, sets the sources and moves them to the appropriate locations. The default is None.

  See *touch_sources()* for details.

- **det** (*nirfasterff.base.optode, optional*) – If not *None*, sets the detectors and moves them to the appropriate locations. The default is None.

  See *touch_detectors()* for details.

- **link** (*int32 NumPy array, optional*) – If not *None*, sets the channel information. Uses one-based indexing. The default is None.

  Each row represents a channel, in the form of *[src, det, active]*, where *active* is 0 or 1

  If *link* contains only two columns, all channels are considered active.

Return type
None.

from_volume(*vol*, *param=utils.MeshingParams()*, *prop=None*, *src=None*, *det=None*, *link=None*)

Construct mesh from a segmented 3D volume using the built-in CGAL mesher. Calls stndmesh.from_solid after meshing step.

Parameters

- **vol** (*uint8 NumPy array*) – 3D segmented volume to be meshed. 0 is considered as outside. Regions labeled using unique integers.

- **param** (nirfasterff.utils.MeshingParams, *optional*) – parameters used in the CGAL mesher. If not specified, uses the default parameters defined in nirfasterff.utils.MeshingParams().

  Please modify fields xPixelSpacing, yPixelSpacing, and SliceThickness if your volume doesn't have [1,1,1] resolution

  See *MeshingParams()* for details.

- **prop** (*double NumPy array, optional*) – If not *None*, calls *dcsmesh.set_prop()* and sets the optical properties in the mesh. The default is None.

  See `set_prop()` for details.

- **src** (*nirfasterff.base.optode, optional*) – If not *None*, sets the sources and moves them to the appropriate locations. The default is None.

  See `touch_sources()` for details.

- **det** (*nirfasterff.base.optode, optional*) – If not *None*, sets the detectors and moves them to the appropriate locations. The default is None.

  See `touch_detectors()` for details.

- **link** (*int32 NumPy array, optional*) – If not *None*, sets the channel information. Uses one-based indexing. The default is None.

  Each row represents a channel, in the form of *[src, det, active]*, where *active* is 0 or 1

  If *link* contains only two columns, all channels are considered active.

Return type
None.

`gen_intmat`(*xgrid*, *ygrid*, *zgrid=[]*)

Calculate the information needed to convert data between mesh and volumetric space, specified by x, y, z (if 3D) grids.

All grids must be uniform. The results will from a nirfasterff.base.meshvol object stored in field .vol

If field .vol already exists, it will be calculated again, and a warning will be thrown

Parameters

- **xgrid** (*double NumPy array*) – x grid in mm.
- **ygrid** (*double NumPy array*) – x grid in mm.
- **zgrid** (*double NumPy array, optional*) – x grid in mm. Leave empty for 2D meshes. The default is [].

Raises
`ValueError` – if grids not uniform, or zgrid empty for 3D mesh

Return type
None.

`isvol`()

Check if convertion matrices between mesh and volumetric spaces are calculated

Returns
True if attribute *.vol* is calculate, False if not.

Return type
bool

`jacobian`(*tvec*, *normalize=True*, *solver=utils.get_solver()*, *opt=utils.SolverOptions()*)

Calculates the Jacobian matrix

Returns the Jacobian, direct field data, and the adjoint data

One Jacobian is calcualted at each time point in tvec, and the derivative is taken with regard to aDb

Parameters

- **tvec** (*double NumPy vector*) – time vector used.
- **normalize** (*bool, optional*) – if True, Jacobbians are normalized to the measured boundary amplitude. The default is True.
- **solver** (*str, optional*) – Choose between 'CPU' or 'GPU' solver (case insensitive). Automatically determined (GPU prioritized) if not specified
- **opt** (nirfasterff.utils.SolverOptions, *optional*) – Solver options. Uses default parameters if not specified, and they should suffice in most cases.

  See *SolverOptions()* for details

Returns

- **J** (*double NumPy array*) – The Jacobian matrix. Size (NChannel, NVoxel, NTime)
- **data1** (*nirfasterff.base.FLdata*) – The calculated direct field. The same as directly calling mesh.femdata(tvec)
- **data2** (*nirfasterff.base.FLdata*) – The calculated adjoint field. The same as calling mesh.femdata(tvec) AFTER swapping the locations of sources and detectors

See also

jacobian_DCS()

save_nirfast(*filename*)

Save mesh in the classic NIRFASTer ASCII format, which is directly compatible with the Matlab version

Parameters
filename (*str*) – name of the file to be saved as. Should have no extensions.

Return type
None.

set_prop(*prop*)

Set optical properties of the whole mesh, using information provided in prop.

Parameters
prop (*double NumPy array*) – optical property info, similar to the MCX format:

```
[region mua(mm-1) musp(mm-1) ri alpha Db]
[region mua(mm-1) musp(mm-1) ri alpha Db]
[...]
```

where 'region' is the region label, and they should match exactly with unique(mesh.region). The order doesn't matter.

Return type
None.

touch_optodes()

Moves all optodes (if non fixed) and recalculate the integration functions (i.e. barycentric coordinates).

See *touch_sources()* and *touch_detectors()* for details

Return type
None.

nirfasterff.base.fluor_mesh

Define the fluorescence mesh class

Classes

fluormesh()	Main class for fluorescence mesh.

nirfasterff.base.fluor_mesh.fluormesh

`class nirfasterff.base.fluor_mesh.`**`fluormesh`**

Bases: `object`

Main class for fluorescence mesh. The methods should cover most of the commonly-used functionalities

`name`

name of the mesh. Default: 'EmptyMesh'

Type
str

`nodes`

locations of nodes in the mesh. Unit: mm. Size (NNodes, dim)

Type
double NumPy array

`bndvtx`

indicator of whether a node is at boundary (1) or internal (0). Size (NNodes,)

Type
double NumPy array

`type`

type of the mesh. It is always 'fluor'.

Type
str

`muax`

intrisic absorption coefficient (mm^-1) at excitation wavelength at each node. Size (NNodes,)

Type
double NumPy array

`muam`

intrisic absorption coefficient (mm^-1) at emission wavelength at each node. Size (NNodes,)

Type
double NumPy array

`muaf`

intrisic absorption coefficient (mm^-1) of the fluorophores at each node. Size (NNodes,)

Type
double NumPy array

`kappax`

intrisic diffusion coefficient (mm) at excitation wavelength at each node. Size (NNodes,). Defined as 1/(3*(muax + musx))

Type
double NumPy array

`kappam`

intrisic diffusion coefficient (mm) at emission wavelength at each node. Size (NNodes,). Defined as 1/(3*(muam + musm))

Type
double NumPy array

`ri`

refractive index at each node. Size (NNodes,)

Type
double NumPy array

`musx`

intrisic reduced scattering coefficient (mm^-1) at excitation wavelength at each node. Size (NNodes,)

Type
double NumPy array

`musm`

intrisic reduced scattering coefficient (mm^-1) at emission wavelength at each node. Size (NNodes,)

Type
double NumPy array

`eta`

quantum efficiency (a.u.) of the fluorophores. Size (NNodes,)

Type
double NumPy array

`tau`

time coefficient (second) of the fluorophores. Size (NNodes,)

Type
double NumPy array

`elements`

triangulation (tetrahedrons or triangles) of the mesh, Size (NElements, dim+1)

Row i contains the indices of the nodes that form tetrahedron/triangle i

One-based indexing for direct interoperatability with the Matlab version

Type
double NumPy array

`region`

region labeling of each node. Starting from 1. Size (NNodes,)

Type
double NumPy array

`source`

information about the sources

Type
nirfasterff.base.optode

`meas`

information about the the detectors

Type
nirfasterff.base.optode

`link`

list of source-detector pairs, i.e. channels. Size (NChannels,3)

First column: source; Second column: detector; Third column: active (1) or not (0)

Type
int32 NumPy array

`c`

light speed (mm/sec) at each node. Size (NNodes,). Defined as c0/ri, where c0 is the light speed in vacuum

Type
double NumPy array

`ksi`

photon fluence rate scale factor on the mesh-outside_mesh boundary as derived from Fresenel's law. Size (NNodes,)

Type
double NumPy array

`element_area`

volume/area (mm^3 or mm^2) of each element. Size (NElements,)

Type
double NumPy array

`support`

total volume/area of all the elements each node belongs to. Size (NNodes,)

Type
double NumPy array

`vol`

object holding information for converting between mesh and volumetric space.

Type
nirfasterff.base.meshvol

`__init__()`

Methods

__init__()	
change_prop(idx, prop)	Change optical properties (muax, musxp, ri, muam, musmp, muaf, eta, and tau) at nodes specified in idx, and automatically change fields kappax, kappam, c, and ksi as well
femdata(freq[, solver, opt, xflag, mmflag, ...])	Calculates fluences for each source using a FEM solver, and then the boudary measurables for each channel
femdata_moments([max_moments, savefield, ...])	Calculates TR moments at each location in the mesh for each source directly using Mellin transform, and then the boudary measurables for each channel
femdata_tpsf(tmax, dt[, savefield, ...])	Calculates TPSF at each location in the mesh for each source using a FEM solver, and then the boudary measurables for each channel
from_copy(mesh)	Deep copy all fields from another mesh.
from_file(file)	Read from classic NIRFAST mesh (ASCII) format, not checking the correctness of the loaded integration functions.
from_mat(matfile[, varname])	Read from Matlab .mat file that contains a NIRFASTer mesh struct.
from_solid(ele, nodes[, prop, src, det, link])	Construct a NIRFASTer mesh from a 3D solid mesh generated by a mesher.
from_volume(vol[, param, prop, src, det, link])	Construct mesh from a segmented 3D volume using the built-in CGAL mesher.
gen_intmat(xgrid, ygrid[, zgrid])	Calculate the information needed to convert data between mesh and volumetric space, specified by x, y, z (if 3D) grids.
isvol()	Check if convertion matrices between mesh and volumetric spaces are calculated
jacobian([freq, normalize, solver, opt])	Calculates the Jacobian matrix
save_nirfast(filename)	Save mesh in the classic NIRFASTer ASCII format, which is directly compatible with the Matlab version
set_prop(prop)	Set optical properties of the whole mesh, using information provided in prop.
touch_optodes()	Moves all optodes (if non fixed) and recalculate the integration functions (i.e. barycentric coordinates).

change_prop(*idx*, *prop*)

Change optical properties (muax, musxp, ri, muam, musmp, muaf, eta, and tau) at nodes specified in idx, and automatically change fields kappax, kappam, c, and ksi as well

Parameters

- **idx** (*list or NumPy array or -1*) – zero-based indices of nodes to change. If *idx==-1*, function changes all the nodes
- **prop** (*list or NumPy array of length 8*) – new optical properties to be assigned to the specified nodes. [muax(mm-1) musxp(mm-1) ri muam(mm-1) musmp(mm-1) muaf(mm-1) eta tau(sec)].

Return type

None.

femdata(*freq*, *solver=utils.get_solver()*, *opt=utils.SolverOptions()*, *xflag=True*, *mmflag=True*, *flflag=True*)

Calculates fluences for each source using a FEM solver, and then the boudary measurables for each channel

If *mesh.vol* is set, fluence data will be represented in volumetric space

See `femdata_fl_CW()` and `femdata_fl_FD()` for details

Parameters

- **freq** (*double*) – modulation frequency in Hz. If CW, set to zero and a more efficient CW solver will be used.
- **solver** (*str, optional*) – Choose between 'CPU' or 'GPU' solver (case insensitive). Automatically determined (GPU prioritized) if not specified
- **opt** (`nirfasterff.utils.SolverOptions`, *optional*) – Solver options. Uses default parameters if not specified, and they should suffice in most cases.

  See `SolverOptions()` for details
- **xflag** (*bool, optional*) – if intrinsic excitation field is calculated. The default is True.
- **mmflag** (*bool, optional*) – if intrinsic emission field is calculated. The default is True.
- **flflag** (*bool, optional*) – if fluorescence field is calculated. If set True, xflag must also be True. The default is True.

Returns

- **data** (*nirfasterff.base.FLdata*) – fluence and boundary measurables given the mesh and optodes.

  See `FLdata()` for details.
- **infox** (*nirfasterff.utils.ConvergenceInfo*) – convergence information of the solver, for intrinsic excitation.

  See `ConvergenceInfo()` for details
- **infom** (*nirfasterff.utils.ConvergenceInfo*) – convergence information of the solver, for intrinsic emission.

  See `ConvergenceInfo()` for details
- **infofl** (*nirfasterff.utils.ConvergenceInfo*) – convergence information of the solver, for fluorescence emission.

  See `ConvergenceInfo()` for details

femdata_moments(*max_moments=3*, *savefield=False*, *solver=utils.get_solver()*, *opt=utils.SolverOptions()*)

Calculates TR moments at each location in the mesh for each source directly using Mellin transform, and then the boudary measurables for each channel

This is done for both excitation and fluorescence emission

If *mesh.vol* is set and *savefield* is set to *True*, internal moments data will be represented in volumetric space

See `femdata_fl_TR_moments()` for details

Parameters

- **max_moments** (*int32, optional*) – max order of moments to calculate. That is, 0th, 1st, 2nd, .., max_moments-th will be calculated. The default is 3.

- **savefield** (*bool, optional*) – If True, the internal moments are also returned. If False, only boundary moments are returned and data.phix and data.phifl will be empty.

  The default is False.

- **solver** (*str, optional*) – Choose between 'CPU' or 'GPU' solver (case insensitive). Automatically determined (GPU prioritized) if not specified

- **opt** (`nirfasterff.utils.SolverOptions`*, optional*) – Solver options. Uses default parameters if not specified, and they should suffice in most cases.

  See *`SolverOptions()`* for details

Returns

- **data** (*nirfasterff.base.flTRMomentsdata*) – internal and boundary moments given the mesh and optodes, both excitation and fluorescence emission.

  See `flTRMomentsdata()` for details.

- **infox** (*nirfasterff.utils.ConvergenceInfo*) – convergence information of the solver, excitation.

  Only the convergence info of highest order moments is returned.

  See *`ConvergenceInfo()`* for details

- **infom** (*nirfasterff.utils.ConvergenceInfo*) – convergence information of the solver, fluorescence emission.

  Only the convergence info of highest order moments is returned.

  See *`ConvergenceInfo()`* for details

femdata_tpsf(*tmax*, *dt*, *savefield=False*, *beautify=True*, *solver=utils.get_solver()*, *opt=utils.SolverOptions()*)

Calculates TPSF at each location in the mesh for each source using a FEM solver, and then the boudary measurables for each channel

This is done for both excitation and fluorescence emission

If *mesh.vol* is set and *savefield* is set to *True*, internal TPSF data will be represented in volumetric space

See *`femdata_fl_TR()`* for details

Parameters

- **tmax** (*double*) – maximum time simulated, in seconds.

- **dt** (*double*) – size of each time step, in seconds.

- **savefield** (*bool, optional*) – If True, the internal TPSFs are also returned. If False, only boundary TPSFs are returned and data.phix and data.phifl will be empty.

  The default is False.

- **beautify** (*bool, optional*) – If true, zeros the initial unstable parts of the boundary TPSFs. The default is True.

- **solver** (*str, optional*) – Choose between 'CPU' or 'GPU' solver (case insensitive). Automatically determined (GPU prioritized) if not specified

- **opt** (`nirfasterff.utils.SolverOptions`*, optional*) – Solver options. Uses default parameters if not specified, and they should suffice in most cases.

  See *`SolverOptions()`* for details

Returns

- **data** (*nirfasterff.base.flTPSFdata*) – internal and boundary TPSFs given the mesh and optodes, both excitation and fluorescence emission.

  See `flTPSFdata()` for details.

- **infox** (*nirfasterff.utils.ConvergenceInfo*) – convergence information of the solver, excitation.

  Only the convergence info of the last time step is returned.

  See `ConvergenceInfo()` for details

- **infom** (*nirfasterff.utils.ConvergenceInfo*) – convergence information of the solver, fluorescence emission.

  Only the convergence info of the last time step is returned.

  See `ConvergenceInfo()` for details

from_copy(*mesh*)

Deep copy all fields from another mesh.

Parameters
mesh (*nirfasterff.base.fluormesh*) – the mesh to copy from.

Return type
None.

from_file(*file*)

Read from classic NIRFAST mesh (ASCII) format, not checking the correctness of the loaded integration functions.

All fields after loading should be directly compatible with Matlab version.

Parameters
file (*str*) – name of the mesh. Any extension will be ignored.

Return type
None.

Examples

```
>>> mesh = nirfasterff.base.fluormesh()
>>> mesh.from_file('meshname')
```

from_mat(*matfile*, *varname=None*)

Read from Matlab .mat file that contains a NIRFASTer mesh struct. All fields copied as is without error checking.

Parameters

- **matfile** (*str*) – name of the .mat file to load. Use of extension is optional.
- **varname** (*str, optional*) – if your .mat file contains multiple variables, use this argument to specify which one to load. The default is None.

  When *varname==None*, *matfile* should contain exatly one structure, which is a NIRFASTer mesh, or the function will do nothing

Return type
None.

`from_solid`(*ele*, *nodes*, *prop=None*, *src=None*, *det=None*, *link=None*)

Construct a NIRFASTer mesh from a 3D solid mesh generated by a mesher. Similar to the solidmesh2nirfast function in Matlab version.

Can also set the optical properties and optodes if supplied

Parameters

- **ele** (*int/double NumPy array*) – element list in one-based indexing. If four columns, all nodes will be labeled as region 1

  If five columns, the last column will be used for region labeling.

- **nodes** (*double NumPy array*) – node locations in the mesh. Unit: mm. Size (NNodes,3).

- **prop** (*double NumPy array, optional*) – If not *None*, calls *fluormesh.set_prop()* and sets the optical properties in the mesh. The default is None.

  See *set_prop()* for details.

- **src** (*nirfasterff.base.optode, optional*) – If not *None*, sets the sources and moves them to the appropriate locations. The default is None.

  See *touch_sources()* for details.

- **det** (*nirfasterff.base.optode, optional*) – If not *None*, sets the detectors and moves them to the appropriate locations. The default is None.

  See *touch_detectors()* for details.

- **link** (*int32 NumPy array, optional*) – If not *None*, sets the channel information. Uses one-based indexing. The default is None.

  Each row represents a channel, in the form of *[src, det, active]*, where *active* is 0 or 1

  If *link* contains only two columns, all channels are considered active.

Return type

None.

`from_volume`(*vol*, *param=utils.MeshingParams()*, *prop=None*, *src=None*, *det=None*, *link=None*)

Construct mesh from a segmented 3D volume using the built-in CGAL mesher. Calls fluormesh.from_solid after meshing step.

Parameters

- **vol** (*uint8 NumPy array*) – 3D segmented volume to be meshed. 0 is considered as outside. Regions labeled using unique integers.

- **param** (`nirfasterff.utils.MeshingParams`, *optional*) – parameters used in the CGAL mesher. If not specified, uses the default parameters defined in nirfasterff.utils.MeshingParams().

  Please modify fields xPixelSpacing, yPixelSpacing, and SliceThickness if your volume doesn't have [1,1,1] resolution

  See *MeshingParams()* for details.

- **prop** (*double NumPy array, optional*) – If not *None*, calls *fluormesh.set_prop()* and sets the optical properties in the mesh. The default is None.

  See *set_prop()* for details.

- **src** (*nirfasterff.base.optode, optional*) – If not *None*, sets the sources and moves them to the appropriate locations. The default is None.

  See `touch_sources()` for details.
- **det** (*nirfasterff.base.optode, optional*) – If not *None*, sets the detectors and moves them to the appropriate locations. The default is None.

  See `touch_detectors()` for details.
- **link** (*int32 NumPy array, optional*) – If not *None*, sets the channel information. Uses one-based indexing. The default is None.

  Each row represents a channel, in the form of *[src, det, active]*, where *active* is 0 or 1

  If *link* contains only two columns, all channels are considered active.

Return type
None.

`gen_intmat`(*xgrid*, *ygrid*, *zgrid=[]*)

Calculate the information needed to convert data between mesh and volumetric space, specified by x, y, z (if 3D) grids.

All grids must be uniform. The results will from a nirfasterff.base.meshvol object stored in field .vol

If field .vol already exists, it will be calculated again, and a warning will be thrown

Parameters

- **xgrid** (*double NumPy array*) – x grid in mm.
- **ygrid** (*double NumPy array*) – x grid in mm.
- **zgrid** (*double NumPy array, optional*) – x grid in mm. Leave empty for 2D meshes. The default is [].

Raises
ValueError – if grids not uniform, or zgrid empty for 3D mesh

Return type
None.

`isvol`()

Check if convertion matrices between mesh and volumetric spaces are calculated

Returns
True if attribute *.vol* is calculate, False if not.

Return type
bool

`jacobian`(*freq=0*, *normalize=True*, *solver=utils.get_solver()*, *opt=utils.SolverOptions()*)

Calculates the Jacobian matrix

Returns the Jacobian, direct field data, and the adjoint data

Structure of the Jacobian is detailed in `jacobian_fl_CW()` and `jacobian_fl_FD()`

Parameters

- **freq** (*double*) – modulation frequency in Hz.
- **normalize** (*bool, optional*) – whether normalize the Jacobian to the amplitudes of boundary measurements at excitation wavelength (‘Born ratio’).

The default is True.

- **solver** (*str, optional*) – Choose between 'CPU' or 'GPU' solver (case insensitive). Automatically determined (GPU prioritized) if not specified
- **opt** (nirfasterff.utils.SolverOptions*, optional*) – Solver options. Uses default parameters if not specified, and they should suffice in most cases.

  See *SolverOptions()* for details

Raises
ValueError – if modulation frequency is negative.

Returns

- **J** (*double NumPy array*) – The Jacobian matrix. Size (NChannel, NVoxel*2) or (NChannel, NVoxel)
- **data1** (*nirfasterff.base.FLdata*) – The calculated direct field. The same as directly calling mesh.femdata(freq)
- **data2** (*nirfasterff.base.FLdata*) – The calculated adjoint field. The same as calling mesh.femdata(freq) AFTER swapping the locations of sources and detectors

save_nirfast(*filename*)

Save mesh in the classic NIRFASTer ASCII format, which is directly compatible with the Matlab version

Parameters
filename (*str*) – name of the file to be saved as. Should have no extensions.

Return type
None.

set_prop(*prop*)

Set optical properties of the whole mesh, using information provided in prop.

Parameters
prop (*double NumPy array*) – optical property info, similar to the MCX format:

```
[region muax(mm-1) musxp(mm-1) ri muam(mm-1) musmp(mm-1) muaf(mm-1)␣
↪eta tau]
[region muax(mm-1) musxp(mm-1) ri muam(mm-1) musmp(mm-1) muaf(mm-1)␣
↪eta tau]
[...]
```

where 'region' is the region label, and they should match exactly with unique(mesh.region). The order doesn't matter.

Return type
None.

touch_optodes()

Moves all optodes (if non fixed) and recalculate the integration functions (i.e. barycentric coordinates).

See *touch_sources()* and *touch_detectors()* for details

Return type
None.

nirfasterff.base.optodes

Define the optode class, an instance of which can be either a source or a detector

Classes

optode([coord])	Class for NIRFASTer optodes, which can be either a group of sources or a group of detectors.

nirfasterff.base.optodes.optode

`class nirfasterff.base.optodes.`**`optode`**(*coord=[]*)

Bases: `object`

Class for NIRFASTer optodes, which can be either a group of sources or a group of detectors.

Note: The field fwhm for sources in the Matlab version has been dropped.

`fixed`

whether an optode is fixed.

If not, it will be moved to one scattering length inside the surface (source) or on the surface (detector).

Default: 0

Type

bool like

`num`

indexing of the optodes, starting from one (1,2,3,…)

Type

double NumPy vector

`coord`

each row is the location of an optode. Unit: mm

Type

double NumPy array

`int_func`

First column is the index (one-based) of the element each optode is in.

The subsequent columns are the barycentric coordinates (i.e. integration function) in the correponding elements. Size (N, dim+2).

Type

double NumPy array

`__init__`(*coord=[]*)

Methods

__init__([coord])	
move_detectors(mesh)	Moves detector to the appropriate locations in the mesh.
move_sources(mesh)	Moves sources to the appropriate locations in the mesh.
touch_detectors(mesh)	Recalculate/fill in all other fields based on 'fixed' and 'coord'.
touch_sources(mesh)	Recalculate/fill in all other fields based on 'fixed' and 'coord'.

move_detectors(*mesh*)

Moves detector to the appropriate locations in the mesh.

For each detector, first move it to the closest point on the surface of the mesh.

Integration functions are NOT calculated after moving, to be consistent with the Matlab version.

Parameters
mesh (*NIRFASTer mesh type*) – The mesh on which the detectors are installed. Can be a 'stndmesh', 'fluormesh', or 'dcsmesh', either 2D or 3D

Raises
ValueError – If mesh.elements does not have 3 or 4 columns, or mesh.dimension is not 2 or 3.

Return type
None.

move_sources(*mesh*)

Moves sources to the appropriate locations in the mesh.

For each source, first move it to the closest point on the surface of the mesh, and then move inside by one scattering length along surface normal.

where scattering length is $1/\mu_s'$ for stnd and dcs mesh, and $1/\mu_{sx}'$ for fluor mesh

Integration functions are also calculated after moving.

Parameters
mesh (*NIRFASTer mesh type*) – The mesh on which the sources are installed. Can be a 'stndmesh', 'fluormesh', or 'dcsmesh', either 2D or 3D

Raises

- **TypeError** – If mesh type is not recognized.
- **ValueError** – If mesh.elements does not have 3 or 4 columns, or mesh.dimension is not 2 or 3.

Return type
None.

touch_detectors(*mesh*)

Recalculate/fill in all other fields based on 'fixed' and 'coord'.

This is useful when a set of detectors are manually added and only the locations are specified.

For non-fixed detectors, function 'move_detectors' is first called, and integration functions are calculated subsequentely.

For fixed detectors, recalculates integration functions directly.

If no detector locations are specified, the function does nothing

Parameters
mesh (*NIRFASTer mesh type*) – The mesh on which the sources are installed. Can be a 'stndmesh', 'fluormesh', or 'dcsmesh', either 2D or 3D

Return type
None.

touch_sources(*mesh*)

Recalculate/fill in all other fields based on 'fixed' and 'coord'.

This is useful when a set of sources are manually added and only the locations are specified.

For non-fixed sources, function 'move_sources' is called, otherwise recalculates integration functions directly

If no source locations are specified, the function does nothing

Parameters
mesh (*NIRFASTer mesh type*) – The mesh on which the sources are installed. Can be a 'stndmesh', 'fluormesh', or 'dcsmesh', either 2D or 3D

Return type
None.

nirfasterff.base.stnd_mesh

Define the standard mesh class

Classes

stndmesh()	Main class for standard mesh.

nirfasterff.base.stnd_mesh.stndmesh

class nirfasterff.base.stnd_mesh.**stndmesh**

Bases: `object`

Main class for standard mesh. The methods should cover most of the commonly-used functionalities

name

name of the mesh. Default: 'EmptyMesh'

Type
str

nodes

locations of nodes in the mesh. Unit: mm. Size (NNodes, dim)

Type
double NumPy array

`bndvtx`

indicator of whether a node is at boundary (1) or internal (0). Size (NNodes,)

Type
double NumPy array

`type`

type of the mesh. It is always ‘stnd’.

Type
str

`mua`

absorption coefficient (mm^-1) at each node. Size (NNodes,)

Type
double NumPy array

`kappa`

diffusion coefficient (mm) at each node. Size (NNodes,). Defined as 1/(3*(mua + mus))

Type
double NumPy array

`ri`

refractive index at each node. Size (NNodes,)

Type
double NumPy array

`mus`

reduced scattering coefficient (mm^-1) at each node. Size (NNodes,)

Type
(double NumPy array

`elements`

triangulation (tetrahedrons or triangles) of the mesh, Size (NElements, dim+1)

Row i contains the indices of the nodes that form tetrahedron/triangle i

One-based indexing for direct interoperatability with the Matlab version

Type
double NumPy array

`region`

region labeling of each node. Starting from 1. Size (NNodes,)

Type
double NumPy array

`source`

information about the sources

Type
nirfasterff.base.optode

`meas`

information about the the detectors

Type
nirfasterff.base.optode

`link`

list of source-detector pairs, i.e. channels. Size (NChannels,3)

First column: source; Second column: detector; Third column: active (1) or not (0)

Type
int32 NumPy array

`c`

light speed (mm/sec) at each node. Size (NNodes,). Defined as c0/ri, where c0 is the light speed in vacuum

Type
double NumPy array

`ksi`

photon fluence rate scale factor on the mesh-outside_mesh boundary as derived from Fresenel's law. Size (NNodes,)

Type
double NumPy array

`element_area`

volume/area (mm^3 or mm^2) of each element. Size (NElements,)

Type
double NumPy array

`support`

total volume/area of all the elements each node belongs to. Size (NNodes,)

Type
double NumPy array

`vol`

object holding information for converting between mesh and volumetric space.

Type
nirfasterff.base.meshvol

`__init__()`

Methods

`__init__`()	
`change_prop`(idx, prop)	Change optical properties (mua, musp, and ri) at nodes specified in idx, and automatically change fields kappa, c, and ksi as well
`femdata`(freq[, solver, opt])	Calculates fluences for each source using a FEM solver, and then the boudary measurables for each channel
`femdata_moments`([max_moments, savefield, ...])	Calculates TR moments at each location in the mesh for each source directly using Mellin transform, and then the boudary measurables for each channel
`femdata_tpsf`(tmax, dt[, savefield, ...])	Calculates TPSF at each location in the mesh for each source using a FEM solver, and then the boudary measurables for each channel
`from_copy`(mesh)	Deep copy all fields from another mesh.
`from_file`(file)	Read from classic NIRFAST mesh (ASCII) format, not checking the correctness of the loaded integration functions.
`from_mat`(matfile[, varname])	Read from Matlab .mat file that contains a NIRFASTer mesh struct.
`from_solid`(ele, nodes[, prop, src, det, link])	Construct a NIRFASTer mesh from a 3D solid mesh generated by a mesher.
`from_volume`(vol[, param, prop, src, det, link])	Construct mesh from a segmented 3D volume using the built-in CGAL mesher.
`gen_intmat`(xgrid, ygrid[, zgrid])	Calculate the information needed to convert data between mesh and volumetric space, specified by x, y, z (if 3D) grids.
`isvol`()	Check if convertion matrices between mesh and volumetric spaces are calculated
`jacobian`([freq, normalize, mus, solver, opt])	Calculates the Jacobian matrix
`save_nirfast`(filename)	Save mesh in the classic NIRFASTer ASCII format, which is directly compatible with the Matlab version
`set_prop`(prop)	Set optical properties of the whole mesh, using information provided in prop.
`touch_optodes`()	Moves all optodes (if non fixed) and recalculate the integration functions (i.e. barycentric coordinates).

change_prop(*idx*, *prop*)

Change optical properties (mua, musp, and ri) at nodes specified in idx, and automatically change fields kappa, c, and ksi as well

Parameters

- **idx** (*list or NumPy array or -1*) – zero-based indices of nodes to change. If *idx==-1*, function changes all the nodes
- **prop** (*list or NumPy array of length 3*) – new optical properties to be assigned to the specified nodes. [mua(mm-1) musp(mm-1) ri].

Return type

None.

femdata(*freq*, *solver=utils.get_solver()*, *opt=utils.SolverOptions()*)

Calculates fluences for each source using a FEM solver, and then the boudary measurables for each channel

If *mesh.vol* is set, fluence data will be represented in volumetric space

See *femdata_stnd_CW()* and *femdata_stnd_FD()* for details

Parameters

- **freq** (*double*) – modulation frequency in Hz. If CW, set to zero and a more efficient CW solver will be used.
- **solver** (*str, optional*) – Choose between 'CPU' or 'GPU' solver (case insensitive). Automatically determined (GPU prioritized) if not specified
- **opt** (nirfasterff.utils.SolverOptions, *optional*) – Solver options. Uses default parameters if not specified, and they should suffice in most cases.

  See *SolverOptions()* for details

Returns

- **data** (*nirfasterff.base.FDdata*) – fluence and boundary measurables given the mesh and optodes.

  See *FDdata()* for details.

- **info** (*nirfasterff.utils.ConvergenceInfo*) – convergence information of the solver.

  See *ConvergenceInfo()* for details

femdata_moments(*max_moments=3*, *savefield=False*, *solver=utils.get_solver()*, *opt=utils.SolverOptions()*)

Calculates TR moments at each location in the mesh for each source directly using Mellin transform, and then the boudary measurables for each channel

If *mesh.vol* is set and *savefield* is set to *True*, internal moments data will be represented in volumetric space

See *femdata_stnd_TR_moments()* for details

Parameters

- **max_moments** (*int32, optional*) – max order of moments to calculate. That is, 0th, 1st, 2nd, .., max_moments-th will be calculated. The default is 3.
- **savefield** (*bool, optional*) – If True, the internal moments are also returned. If False, only boundary moments are returned and data.phi will be empty.

  The default is False.

- **solver** (*str, optional*) – Choose between 'CPU' or 'GPU' solver (case insensitive). Automatically determined (GPU prioritized) if not specified
- **opt** (nirfasterff.utils.SolverOptions, *optional*) – Solver options. Uses default parameters if not specified, and they should suffice in most cases.

  See *SolverOptions()* for details

Returns

- **data** (*nirfasterff.base.TRMomentsdata*) – internal and boundary moments given the mesh and optodes.

  See *TRMomentsdata()* for details.

- **info** (*nirfasterff.utils.ConvergenceInfo*) – convergence information of the solver.

  Only the convergence info of highest order moments is returned.

See `ConvergenceInfo()` for details

`femdata_tpsf`(*tmax*, *dt*, *savefield=False*, *beautify=True*, *solver=utils.get_solver()*, *opt=utils.SolverOptions()*)

Calculates TPSF at each location in the mesh for each source using a FEM solver, and then the boudary measurables for each channel

If *mesh.vol* is set and *savefield* is set to *True*, internal TPSF data will be represented in volumetric space

See *`femdata_stnd_TR()`* for details

Parameters

- **`tmax`** (*`double`*) – maximum time simulated, in seconds.
- **`dt`** (*`double`*) – size of each time step, in seconds.
- **`savefield`** (*`bool, optional`*) – If True, the internal TPSFs are also returned. If False, only boundary TPSFs are returned and data.phi will be empty.

  The default is False.
- **`beautify`** (*`bool, optional`*) – If true, zeros the initial unstable parts of the boundary TPSFs. The default is True.
- **`solver`** (*`str, optional`*) – Choose between 'CPU' or 'GPU' solver (case insensitive). Automatically determined (GPU prioritized) if not specified
- **`opt`** (`nirfasterff.utils.SolverOptions`, *`optional`*) – Solver options. Uses default parameters if not specified, and they should suffice in most cases.

  See *`SolverOptions()`* for details

Returns

- **data** (*nirfasterff.base.TPSFdata*) – internal and boundary TPSFs given the mesh and optodes.

  See *`TPSFdata()`* for details.
- **info** (*nirfasterff.utils.ConvergenceInfo*) – convergence information of the solver.

  Only the convergence info of the last time step is returned.

  See `ConvergenceInfo()` for details

`from_copy`(*mesh*)

Deep copy all fields from another mesh.

Parameters
`mesh` (*`nirfasterff.base.stndmesh`*) – the mesh to copy from.

Return type
None.

`from_file`(*file*)

Read from classic NIRFAST mesh (ASCII) format, not checking the correctness of the loaded integration functions.

All fields after loading should be directly compatible with Matlab version.

Parameters
`file` (*`str`*) – name of the mesh. Any extension will be ignored.

Return type
None.

Examples

```
>>> mesh = nirfasterff.base.stndmesh()
>>> mesh.from_file('meshname')
```

from_mat(*matfile*, *varname=None*)

Read from Matlab .mat file that contains a NIRFASTer mesh struct. All fields copied as is without error checking.

Parameters

- **matfile** (*str*) – name of the .mat file to load. Use of extension is optional.
- **varname** (*str, optional*) – if your .mat file contains multiple variables, use this argument to specify which one to load. The default is None.

  When *varname==None*, *matfile* should contain exatly one structure, which is a NIRFASTer mesh, or the function will do nothing

Return type

None.

from_solid(*ele*, *nodes*, *prop=None*, *src=None*, *det=None*, *link=None*)

Construct a NIRFASTer mesh from a 3D solid mesh generated by a mesher. Similar to the solidmesh2nirfast function in Matlab version.

Can also set the optical properties and optodes if supplied

Parameters

- **ele** (*int/double NumPy array*) – element list in one-based indexing. If four columns, all nodes will be labeled as region 1

  If five columns, the last column will be used for region labeling.
- **nodes** (*double NumPy array*) – node locations in the mesh. Unit: mm. Size (NNodes,3).
- **prop** (*double NumPy array, optional*) – If not *None*, calls *stndmesh.set_prop()* and sets the optical properties in the mesh. The default is None.

  See *set_prop()* for details.
- **src** (*nirfasterff.base.optode, optional*) – If not *None*, sets the sources and moves them to the appropriate locations. The default is None.

  See *touch_sources()* for details.
- **det** (*nirfasterff.base.optode, optional*) – If not *None*, sets the detectors and moves them to the appropriate locations. The default is None.

  See *touch_detectors()* for details.
- **link** (*int32 NumPy array, optional*) – If not *None*, sets the channel information. Uses one-based indexing. The default is None.

  Each row represents a channel, in the form of *[src, det, active]*, where *active* is 0 or 1

  If *link* contains only two columns, all channels are considered active.

Return type

None.

from_volume(*vol*, *param=utils.MeshingParams()*, *prop=None*, *src=None*, *det=None*, *link=None*)

Construct mesh from a segmented 3D volume using the built-in CGAL mesher. Calls stndmesh.from_solid after meshing step.

Parameters

- **vol** (*uint8 NumPy array*) – 3D segmented volume to be meshed. 0 is considered as outside. Regions labeled using unique integers.
- **param** (`nirfasterff.utils.MeshingParams`, *optional*) – parameters used in the CGAL mesher. If not specified, uses the default parameters defined in nirfasterff.utils.MeshingParams().

  Please modify fields xPixelSpacing, yPixelSpacing, and SliceThickness if your volume doesn't have [1,1,1] resolution

  See *MeshingParams()* for details.
- **prop** (*double NumPy array, optional*) – If not *None*, calls *stndmesh.set_prop()* and sets the optical properties in the mesh. The default is None.

  See *set_prop()* for details.
- **src** (*nirfasterff.base.optode, optional*) – If not *None*, sets the sources and moves them to the appropriate locations. The default is None.

  See *touch_sources()* for details.
- **det** (*nirfasterff.base.optode, optional*) – If not *None*, sets the detectors and moves them to the appropriate locations. The default is None.

  See *touch_detectors()* for details.
- **link** (*int32 NumPy array, optional*) – If not *None*, sets the channel information. Uses one-based indexing. The default is None.

  Each row represents a channel, in the form of *[src, det, active]*, where *active* is 0 or 1

  If *link* contains only two columns, all channels are considered active.

Return type

None.

gen_intmat(*xgrid*, *ygrid*, *zgrid=[]*)

Calculate the information needed to convert data between mesh and volumetric space, specified by x, y, z (if 3D) grids.

All grids must be uniform. The results will from a nirfasterff.base.meshvol object stored in field .vol

If field .vol already exists, it will be calculated again, and a warning will be thrown

Parameters

- **xgrid** (*double NumPy array*) – x grid in mm.
- **ygrid** (*double NumPy array*) – x grid in mm.
- **zgrid** (*double NumPy array, optional*) – x grid in mm. Leave empty for 2D meshes. The default is [].

Raises

ValueError – if grids not uniform, or zgrid empty for 3D mesh

Return type

None.

`isvol()`

Check if convertion matrices between mesh and volumetric spaces are calculated

Returns

True if attribute *.vol* is calculate, False if not.

Return type

bool

`jacobian`(*freq=0*, *normalize=True*, *mus=False*, *solver=utils.get_solver()*, *opt=utils.SolverOptions()*)

Calculates the Jacobian matrix

Returns the Jacobian, direct field data, and the adjoint data

Structure of the Jacobian is detailed in `jacobian_stnd_CW()` and `jacobian_stnd_FD()`

Parameters

- **freq** (*double*) – modulation frequency, in Hz.
- **normalize** (*bool, optional*) – whether normalize the Jacobian to the amplitudes of boundary measurements, i.e. use Rytov approximation.

  The default is True.
- **mus** (*bool, optional*) – whether derivates wrt mus (left half of the 'full' Jacobian) is calculated. Only has effect when freq=0. The default is True.
- **solver** (*str, optional*) – Choose between 'CPU' or 'GPU' solver (case insensitive). Automatically determined (GPU prioritized) if not specified
- **opt** (`nirfasterff.utils.SolverOptions`, *optional*) – Solver options. Uses default parameters if not specified, and they should suffice in most cases.

  See `SolverOptions()` for details

Raises

ValueError – if freq is negative.

Returns

- **J** (*double or complex double NumPy array*) – The Jacobian matrix. Size (NChannel*2, NVoxel*2) if freq>0, (NChannel, NVoxel*2) if freq=0 and mus=True, (NChannel, NVoxel) if freq=0 and mus=False
- **data1** (*nirfasterff.base.FDdata*) – The calculated direct field. The same as directly calling mesh.femdata(freq)
- **data2** (*nirfasterff.base.FDdata*) – The calculated adjoint field. The same as calling mesh.femdata(freq) AFTER swapping the locations of sources and detectors

`save_nirfast`(*filename*)

Save mesh in the classic NIRFASTer ASCII format, which is directly compatible with the Matlab version

Parameters

filename (*str*) – name of the file to be saved as. Should have no extensions.

Return type

None.

`set_prop`(*prop*)

Set optical properties of the whole mesh, using information provided in prop.

Parameters

prop (*double NumPy array*) – optical property info, similar to the MCX format:

```
[region mua(mm-1) musp(mm-1) ri]
[region mua(mm-1) musp(mm-1) ri]
[...]
```

where 'region' is the region label, and they should match exactly with unique(mesh.region). The order doesn't matter.

Return type
None.

touch_optodes()

Moves all optodes (if non fixed) and recalculate the integration functions (i.e. barycentric coordinates).

See *touch_sources()* and *touch_detectors()* for details

Return type
None.

4.1.2 nirfasterff.forward

Funtions for forward data calculation

Modules

analytical	Analytical solutions to the diffusion equation in semi-infinite media
femdata	The FEM solvers.

nirfasterff.forward.analytical

Analytical solutions to the diffusion equation in semi-infinite media

Functions

semi_infinite_CW(mua, musp, n, rho[, z, ...])	Calculates the continuous-wave fluence in space using the analytical solution to the diffusion equation in semi-infinite media.
semi_infinite_DCS(mua, musp, n, rho, aDb, ...)	Calculates DCS G1 curve using the analytical solution to the correlation diffusion equation in semi-infinite media
semi_infinite_FD(mua, musp, n, freq, rho[, ...])	Calculates the frequency-domain fluence in space using the analytical solution to the diffusion equation in semi-infinite media.
semi_infinite_TR(mua, musp, n, rho, T, dt[, ...])	Calculates TPSF at a given location using the analytical solution to the diffusion equation in semi-infinite media

nirfasterff.forward.analytical.semi_infinite_CW

nirfasterff.forward.analytical.**semi_infinite_CW**(*mua*, *musp*, *n*, *rho*, *z=0*, *boundary='exact'*, *n_air=1.0*)

Calculates the continuous-wave fluence in space using the analytical solution to the diffusion equation in semi-infinite media.

Internaly calls the FD version with freq set to zero

Parameters

- **mua** (*double*) – absorption coefficient of the medium, in mm^-1.
- **musp** (*double*) – reduced scattering coefficient of the medium, in mm^-1.
- **n** (*double*) – refractive index of the medium.
- **rho** (*double NumPy vector or scalar*) – distance to the light source, projected to the x-y (i.e. boundary plane of the semi-infinite space) plane, in mm.

  Can be a vector, in which case fluences calculated at multiple locations will be returned
- **z** (*double NumPy vector or scalar, optional*) – depth of the location(s) of interest. 0 for boundary measurement.

  If a vector, it must have the same length as rho

  The default is 0.
- **boundary** (*str, optional*) – type of the boundary condition, which can be 'robin', 'approx', or 'exact'. The default is 'exact'.

  See *boundary_attenuation()* for details.
- **n_air** (*double, optional*) – refratcive index outside of the semi-infinite space, which is typically assumed to be air. The default is 1.0.

Returns

calculated fluence, where each element corresponds to a location, as specified by rho and z. Size (NLocation,)

Return type

double NumPy vector

References

Durduran et al, 2010, Rep. Prog. Phys. doi:10.1088/0034-4885/73/7/076701

nirfasterff.forward.analytical.semi_infinite_DCS

nirfasterff.forward.analytical.**semi_infinite_DCS**(*mua*, *musp*, *n*, *rho*, *aDb*, *wvlength*, *tvec*, *z=0*, *boundary='exact'*, *n_air=1.0*, *normalize=0*)

Calculates DCS G1 curve using the analytical solution to the correlation diffusion equation in semi-infinite media

Function assumes Brownian motion, that is, $\langle \Delta r^2 \rangle = 6\alpha Db\tau$

Parameters

- **mua** (*double*) – absorption coefficient of the medium, in mm^-1.
- **musp** (*double*) – reduced scattering coefficient of the medium, in mm^-1.
- **n** (*double*) – refractive index of the medium.
- **rho** (*scalar*) – distance to the light source, projected to the x-y (i.e. boundary plane of the semi-infinite space) plane, in mm.
- **aDb** (*double*) – The lumped flow parameter αDb in Brownian motion.
- **wvlength** (*double or int*) – wavelength used, in nm.
- **tvec** (*double Numpy vector*) – time vector used to calculate the G1 curve. It is usually a good idea to use log space.

- **z** (*double, optional*) – depth of the location of interest. 0 for boundary measurement. The default is 0.
- **boundary** (*str, optional*) – type of the boundary condition, which can be 'robin', 'approx', or 'exact'. The default is 'exact'.

  See `boundary_attenuation()` for details.
- **n_air** (*double, optional*) – refratcive index outside of the semi-infinite space, which is typically assumed to be air. The default is 1.0.
- **normalize** (*bool, optional*) – if true, returns the normalized g1 curve, instead of G1. The default is 0.

Returns
G1 or g1 curve calculated at the given location and time points.

Return type
double NumPy vector

References

Durduran et al, 2010, Rep. Prog. Phys. doi:10.1088/0034-4885/73/7/076701

nirfasterff.forward.analytical.semi_infinite_FD

`nirfasterff.forward.analytical.`**`semi_infinite_FD`**`(`*mua*, *musp*, *n*, *freq*, *rho*, *z=0*, *boundary='exact'*, *n_air=1.0*`)`

Calculates the frequency-domain fluence in space using the analytical solution to the diffusion equation in semi-infinite media.

Parameters

- **mua** (*double*) – absorption coefficient of the medium, in mm^-1.
- **musp** (*double*) – reduced scattering coefficient of the medium, in mm^-1.
- **n** (*double*) – refractive index of the medium.
- **freq** (*double NumPy vector or scalar*) – modulation frequency, in Hz.
- **rho** (*double NumPy vector or scalar*) – distance to the light source, projected to the x-y (i.e. boundary plane of the semi-infinite space) plane, in mm.

  Can be a vector, in which case fluences calculated at multiple locations will be returned
- **z** (*double NumPy vector or scalar, optional*) – depth of the location(s) of interest. 0 for boundary measurement.

  If a vector, it must have the same length as rho

  The default is 0.
- **boundary** (*str, optional*) – type of the boundary condition, which can be 'robin', 'approx', or 'exact'. The default is 'exact'.

  See `boundary_attenuation()` for details.
- **n_air** (*double, optional*) – refratcive index outside of the semi-infinite space, which is typically assumed to be air. The default is 1.0.

Returns

calculated complex fluence, where each row (or element in vector) corresponds to a location, as specified by rho and z,

and each column corresponds to a modulation frequency. Size (NLocation,), or (NLocation, NFreq)

Return type
complex double NumPy array

References

Durduran et al, 2010, Rep. Prog. Phys. doi:10.1088/0034-4885/73/7/076701

nirfasterff.forward.analytical.semi_infinite_TR

`nirfasterff.forward.analytical.`**`semi_infinite_TR`**(*mua*, *musp*, *n*, *rho*, *T*, *dt*, *z=0*, *boundary='EBC-Robin'*)

Calculates TPSF at a given location using the analytical solution to the diffusion equation in semi-infinite media

Parameters

- **mua** (*double*) – absorption coefficient of the medium, in mm^-1.
- **musp** (*double*) – reduced scattering coefficient of the medium, in mm^-1.
- **n** (*double*) – refractive index of the medium.
- **rho** (*scalar*) – distance to the light source, projected to the x-y (i.e. boundary plane of the semi-infinite space) plane, in mm.
- **T** (*double NumPy vector or scalar*) – if a scalar, it is the total amount of time and time vector will be generated also based on dt (see below).

  if a vector, it is directly used as the time vector, and the argument dt will be ignored. Unit: seconds
- **dt** (*double*) – step size of the time vector (in seconds). If the argument T is a vector, it will be ignored.
- **z** (*double, optional*) – depth of the location of interest. 0 for boundary measurement. The default is 0.

  Note that when *z=0*, the function returns reflectance, instead of fluence. Please refer to the References for detail.
- **boundary** (*str, optional*) – type of the boundary condition, which can be (case insensitive),

  'PCB-exact' - partial current boundary condition, with exact internal reflectance

  'PCB-approx' - partial current boundary condition, with Groenhuis internal reflectance approximation

  'PCB-Robin' - partial current boundary condition, with internal reflectance derived from Fresnel's law

  'EBC-exact' - extrapolated boundary condition, with exact internal reflectance

  'EBC-approx' - extrapolated boundary condition, with Groenhuis internal reflectance approximation

  'EBC-Robin' - extrapolated boundary condition, with internal reflectance derived from Fresnel's law

’ZBC’ - zero boundary condition

The default is ‘EBC-Robin’.

See `boundary_attenuation()` for the differences between ‘exact’, ‘approx’ and ‘robin’

Raises
`ValueError` – if boundary condition is not of a recognized kind.

Returns
phi – Coumn 0: the time vector; Column 1: calculated TPSF at the given location. Size (NTime, 2)

Return type
double Numpy array

References

Hielscher et al., 1995, Phys. Med. Biol. doi:10.1088/0031-9155/40/11/013

Kienle and Patterson, 1997, JOSA A. doi:10.1364/JOSAA.14.000246

nirfasterff.forward.femdata

The FEM solvers. It is usually recommended to use the high-level wrappers in the mesh classes by calling the mesh.femdata* functions.

Functions

femdata_DCS(mesh, tvec[, solver, opt])	Forward modeling calculating steady-state fluences and G1/g1 curves by solving the correlation diffusion equation.
femdata_fl_CW(mesh[, solver, opt, xflag, ...])	Forward modeling for CW in fluorescence meshes.
femdata_fl_FD(mesh, freq[, solver, opt, ...])	Forward modeling for FD in fluorescence meshes.
femdata_fl_TR(mesh, tmax, dt[, savefield, ...])	Forward modeling calculating TPSF on a fluorescence mesh.
femdata_fl_TR_moments(mesh[, max_moments, ...])	Forward modeling calculating TR moments using Mellin transform on a fluorescence mesh.
femdata_stnd_CW(mesh[, solver, opt])	Forward modeling for CW.
femdata_stnd_FD(mesh, freq[, solver, opt])	Forward modeling for FD.
femdata_stnd_TR(mesh, tmax, dt[, savefield, ...])	Forward modeling calculating TPSF on a standard mesh.
femdata_stnd_TR_moments(mesh[, max_moments, ...])	Forward modeling calculating TR moments using Mellin transform on a standard mesh.

nirfasterff.forward.femdata.femdata_DCS

nirfasterff.forward.femdata.**femdata_DCS**(*mesh*, *tvec*, *solver=utils.get_solver()*, *opt=utils.SolverOptions()*)

Forward modeling calculating steady-state fluences and G1/g1 curves by solving the correlation diffusion equation. Please consider using mesh.femdata(tvec) instead.

The fluences as well as its boundary amplitudes are exactly the same as what the CW solver would give when tau=0.

g1 curve is simple G1 curve normalized by the boundary amplitudes.

The function calculates the MASS matrices, the source vectors, and calls the G1 solver, which internally utilizes the CW solver.

When calculating the flow-related term, the function assumes Brownian motion and uses only mesh.aDb (that is, mesh.a and mesh.Db are ignored).

This is to say, we assume $\langle \Delta r^2 \rangle = 6\alpha Db\tau$

Parameters

- **mesh** (*nirfasterff.base.dcsmesh*) – the mesh used to calcuate the forward data.
- **tvec** (*double NumPy array*) – time vector (i.e. :math:` au`) for the G1 curve, in seconds. It is usually a good idea to use log scale
- **solver** (*str, optional*) – Choose between 'CPU' or 'GPU' solver (case insensitive). Automatically determined (GPU prioritized) if not specified
- **opt** (nirfasterff.utils.SolverOptions*, optional*) – Solver options. Uses default parameters if not specified, and they should suffice in most cases.

  See *SolverOptions()* for details

Raises

TypeError – if mesh is not a dcs mesh.

Returns

- **data** (*nirfasterff.base.DCSdata*) – contains fluence, G1 curve, and g1 curve calculated at each spatial location, and also the boundary data.

  If mesh.vol is set, internal G1 curves will be returned in volumetric space

  See *DCSdata()* for details.
- **info** (*nirfasterff.utils.ConvergenceInfo*) – convergence information of the solver when calculating the fluence field.

  See *ConvergenceInfo()* for details

See also

gen_mass_matrix(), *gen_sources()*, and *get_field_CW()*

References

Durduran et al, 2010, Rep. Prog. Phys. doi:10.1088/0034-4885/73/7/076701

nirfasterff.forward.femdata.femdata_fl_CW

nirfasterff.forward.femdata.**femdata_fl_CW**(*mesh*, *solver=utils.get_solver()*, *opt=utils.SolverOptions()*, *xflag=True*, *mmflag=True*, *flflag=True*)

Forward modeling for CW in fluorescence meshes. Please consider using mesh.femdata(0) instead.

The function calculates the MASS matrices, the source vectors, and calls the CW solver (preconditioned conjugated gradient).

The optional flags can be used to determine which fields are calculated. By default all true.

Note that when flflag is set to True, xflag must also be True.

Parameters

- **mesh** (*nirfasterff.base.fluormesh*) – the mesh used to calcuate the forward data.

- **solver** (*str, optional*) – Choose between 'CPU' or 'GPU' solver (case insensitive). Automatically determined (GPU prioritized) if not specified
- **opt** (nirfasterff.utils.SolverOptions, *optional*) – Solver options. Uses default parameters if not specified, and they should suffice in most cases.

  See *SolverOptions()* for details
- **xflag** (*bool, optional*) – if intrinsic excitation field is calculated. The default is True.
- **mmflag** (*bool, optional*) – if intrinsic emission field is calculated. The default is True.
- **flflag** (*bool, optional*) – if fluorescence field is calculated. If set True, xflag must also be True. The default is True.

Raises

- **TypeError** – If mesh is not a fluor mesh.
- **ValueError** – If flflag is set True but xflag is not.

Returns

- **data** (*nirfasterff.base.FLdata*) – fluence and boundary measurables given the mesh and optodes.

  If mesh.vol is defined, the returned fluences will be in volumetric space

  See *FLdata()* for details.
- **infox** (*nirfasterff.utils.ConvergenceInfo*) – convergence information of the solver, for intrinsic excitation.

  See *ConvergenceInfo()* for details
- **infom** (*nirfasterff.utils.ConvergenceInfo*) – convergence information of the solver, for intrinsic emission.

  See *ConvergenceInfo()* for details
- **infofl** (*nirfasterff.utils.ConvergenceInfo*) – convergence information of the solver, for fluorescence emission.

  See *ConvergenceInfo()* for details

See also

get_field_CW(), *gen_mass_matrix()*, *gen_sources()*, and *gen_sources_fl()*

nirfasterff.forward.femdata.femdata_fl_FD

nirfasterff.forward.femdata.**femdata_fl_FD**(*mesh*, *freq*, *solver=utils.get_solver()*, *opt=utils.SolverOptions()*, *xflag=True*, *mmflag=True*, *flflag=True*)

Forward modeling for FD in fluorescence meshes. Please consider using mesh.femdata(frequency) instead.

The function calculates the MASS matrix, the source vectors, and calls the FD solver (preconditioned BiCGStab).

The optional flags can be used to determine which fields are calculated. By default all true.

Note that when flflag is set to True, xflag must also be True.

Parameters

- **mesh** (*nirfasterff.base.stndmesh*) – the mesh used to calcuate the forward data.
- **freq** (*double*) – modulation frequency in Hz.

  When it is 0, function continues with the BiCGstab solver, but generates a warning that the CW solver should be used for better performance
- **solver** (*str, optional*) – Choose between 'CPU' or 'GPU' solver (case insensitive). Automatically determined (GPU prioritized) if not specified
- **opt** (nirfasterff.utils.SolverOptions, *optional*) – Solver options. Uses default parameters if not specified, and they should suffice in most cases.

  See *SolverOptions()* for details
- **xflag** (*bool, optional*) – if intrinsic excitation field is calculated. The default is True.
- **mmflag** (*bool, optional*) – if intrinsic emission field is calculated. The default is True.
- **flflag** (*bool, optional*) – if fluorescence field is calculated. If set True, xflag must also be True. The default is True.

Raises

- **TypeError** – If mesh is not a fluor mesh.
- **ValueError** – If flflag is set True but xflag is not.

Returns

- **data** (*nirfasterff.base.FLdata*) – fluence and boundary measurables given the mesh and optodes.

  If mesh.vol is defined, the returned fluences will be in volumetric space

  See *FLdata()* for details.
- **infox** (*nirfasterff.utils.ConvergenceInfo*) – convergence information of the solver, for intrinsic excitation.

  See *ConvergenceInfo()* for details
- **infom** (*nirfasterff.utils.ConvergenceInfo*) – convergence information of the solver, for intrinsic emission.

  See *ConvergenceInfo()* for details
- **infofl** (*nirfasterff.utils.ConvergenceInfo*) – convergence information of the solver, for fluorescence emission.

  See *ConvergenceInfo()* for details

See also

get_field_FD(), *gen_mass_matrix()*, *gen_sources()*, and *gen_sources_fl()*

nirfasterff.forward.femdata.femdata_fl_TR

nirfasterff.forward.femdata.**femdata_fl_TR**(*mesh*, *tmax*, *dt*, *savefield=False*, *beautify=True*, *solver=utils.get_solver()*, *opt=utils.SolverOptions()*)

Forward modeling calculating TPSF on a fluorescence mesh. Please consider using mesh.femdata_tpsf(tmax, dt) instead.

The function calculates the MASS matrices, the source vectors, and calls two separate TPSF solvers (both preconditioned conjugated gradient):

First time calculates the TPSF for the excitation field, the result of which is consequently convolved with the decay.

The second solver is called with the convolved excitation field as its input to calculate the TPSF for fluorescence emission.

Parameters

- **mesh** (*nirfasterff.base.fluormesh*) – the mesh used to calcuate the forward data.
- **tmax** (*double*) – maximum time simulated, in seconds.
- **dt** (*double*) – size of each time step, in seconds.
- **savefield** (*bool, optional*) – If True, the internal TPSFs are also returned. If False, only boundary TPSFs are returned and data.phix and data.phifl will be empty.

  The default is False.
- **beautify** (*bool, optional*) – If true, zeros the initial unstable parts of the boundary TPSFs. The default is True.
- **solver** (*str, optional*) – Choose between 'CPU' or 'GPU' solver (case insensitive). Automatically determined (GPU prioritized) if not specified
- **opt** (nirfasterff.utils.SolverOptions, *optional*) – Solver options. Uses default parameters if not specified, and they should suffice in most cases.

  See *SolverOptions()* for details

Raises

TypeError – if mesh is not a fluor mesh.

Returns

- **data** (*nirfasterff.base.flTPSFdata*) – internal and boundary TPSFs given the mesh and optodes, both excitation and fluorescence emission.

  If *mesh.vol* is set and *savefield* is set to *True*, internal TPSF data will be represented in volumetric space

  See *flTPSFdata()* for details.
- **infox** (*nirfasterff.utils.ConvergenceInfo*) – convergence information of the solver, excitation.

  Only the convergence info of the last time step is returned.

  See *ConvergenceInfo()* for details
- **infom** (*nirfasterff.utils.ConvergenceInfo*) – convergence information of the solver, fluorescence emission.

  Only the convergence info of the last time step is returned.

  See *ConvergenceInfo()* for details

See also

gen_mass_matrix(), *get_field_TR()*, and *get_field_TRFL()*

nirfasterff.forward.femdata.femdata_fl_TR_moments

`nirfasterff.forward.femdata.`**`femdata_fl_TR_moments`**`(`*mesh*, *max_moments=3*, *savefield=False*, *solver=utils.get_solver()*, *opt=utils.SolverOptions()*`)`

Forward modeling calculating TR moments using Mellin transform on a fluorescence mesh. Please consider using mesh.femdata_moments() instead.

The function calculates the MASS matrix, the source vectors, and calls the two Mellin moments solver (both preconditioned conjugated gradient):

First time calculates the moments for the excitation field, the result of which is consequently used as the input of the second solver,

which calculates the moments of fluorescence emission based on the excitation moments

Calculates 0th, 1st, 2nd, .., max_moments-th moments directly without calculating TPSF first, and this is done for both excitation and fluorescence emission.

This is more efficient, if the time series are not of concern.

Parameters

- **mesh** (*nirfasterff.base.fluormesh*) – the mesh used to calculate the forward data.
- **max_moments** (*int32, optional*) – max order of moments to calculate. That is, 0th, 1st, 2nd, .., max_moments-th will be calculated. The default is 3.
- **savefield** (*bool, optional*) – If True, the internal moments are also returned. If False, only boundary moments are returned and data.phix and data.phifl will be empty.

  The default is False.
- **solver** (*str, optional*) – Choose between 'CPU' or 'GPU' solver (case insensitive). Automatically determined (GPU prioritized) if not specified
- **opt** (`nirfasterff.utils.SolverOptions`*, optional*) – Solver options. Uses default parameters if not specified, and they should suffice in most cases.

  See *SolverOptions()* for details

Raises

- **TypeError** – if mesh is not a fluor mesh.
- **ValueError** – if max_moments is negative.

Returns

- **data** (*nirfasterff.base.flTRMomentsdata*) – internal and boundary moments given the mesh and optodes, both excitation and fluorescence emission.

  If *mesh.vol* is set and *savefield* is set to *True*, internal moments will be represented in volumetric space

  See *flTRMomentsdata()* for details.
- **infox** (*nirfasterff.utils.ConvergenceInfo*) – convergence information of the solver, excitation.

  Only the convergence info of highest order moments is returned.

  See *ConvergenceInfo()* for details
- **infom** (*nirfasterff.utils.ConvergenceInfo*) – convergence information of the solver, fluorescence emission.

Only the convergence info of highest order moments is returned.

See `ConvergenceInfo()` for details

See also

gen_mass_matrix(), *get_field_TRmoments()*, and *get_field_TRFLmoments()*

nirfasterff.forward.femdata.femdata_stnd_CW

`nirfasterff.forward.femdata.`**`femdata_stnd_CW`**(*mesh*, *solver=utils.get_solver()*, *opt=utils.SolverOptions()*)

Forward modeling for CW. Please consider using mesh.femdata(0) instead.

The function calculates the FEM MASS matrix, the source vectors, and calls the CW solver (preconditioned conjugated gradient).

Parameters

- **mesh** (*nirfasterff.base.stndmesh*) – the mesh used to calcuate the forward data.
- **solver** (*str, optional*) – Choose between 'CPU' or 'GPU' solver (case insensitive). Automatically determined (GPU prioritized) if not specified
- **opt** (*nirfasterff.utils.SolverOptions, optional*) – Solver options. Uses default parameters if not specified, and they should suffice in most cases.

  See `SolverOptions()` for details

Raises

TypeError – If mesh is not a stnd mesh.

Returns

- **data** (*nirfasterff.base.FDdata*) – fluence and boundary measurables given the mesh and optodes.

  If mesh.vol is defined, the returned fluence will be in volumetric space

  See `FDdata()` for details.
- **info** (*nirfasterff.utils.ConvergenceInfo*) – convergence information of the solver.

  See `ConvergenceInfo()` for details

See also

get_field_CW(), *gen_mass_matrix()*, and *gen_sources()*

nirfasterff.forward.femdata.femdata_stnd_FD

`nirfasterff.forward.femdata.`**`femdata_stnd_FD`**(*mesh*, *freq*, *solver=utils.get_solver()*, *opt=utils.SolverOptions()*)

Forward modeling for FD. Please consider using mesh.femdata(freq) instead. freq in Hz

The function calculates the MASS matrix, the source vectors, and calls the FD solver (preconditioned BiCGStab).

Parameters

- **mesh** (*nirfasterff.base.stndmesh*) – the mesh used to calcuate the forward data.

- **freq** (*double*) – modulation frequency in Hz.

  When it is 0, function continues with the BiCGstab solver, but generates a warning that the CW solver should be used for better performance

- **solver** (*str, optional*) – Choose between 'CPU' or 'GPU' solver (case insensitive). Automatically determined (GPU prioritized) if not specified

- **opt** (`nirfasterff.utils.SolverOptions`, *optional*) – Solver options. Uses default parameters if not specified, and they should suffice in most cases.

  See *SolverOptions()* for details

Raises

TypeError – If mesh is not a stnd mesh.

Returns

- **data** (*nirfasterff.base.FDdata*) – fluence and boundary measurables given the mesh and optodes.

  If mesh.vol is defined, the returned fluence will be in volumetric space

  See *FDdata()* for details.

- **info** (*nirfasterff.utils.ConvergenceInfo*) – convergence information of the solver.

  See *ConvergenceInfo()* for details

See also

get_field_FD(), *gen_mass_matrix()*, and *gen_sources()*

nirfasterff.forward.femdata.femdata_stnd_TR

nirfasterff.forward.femdata.**femdata_stnd_TR**(*mesh*, *tmax*, *dt*, *savefield=False*, *beautify=True*, *solver=utils.get_solver()*, *opt=utils.SolverOptions()*)

Forward modeling calculating TPSF on a standard mesh. Please consider using mesh.femdata_tpsf(tmax, dt) instead.

The function calculates the MASS matrices, the source vectors, and calls the standard TPSF solver (preconditioned conjugated gradient).

Parameters

- **mesh** (*nirfasterff.base.stndmesh*) – the mesh used to calcuate the forward data.

- **tmax** (*double*) – maximum time simulated, in seconds.

- **dt** (*double*) – size of each time step, in seconds.

- **savefield** (*bool, optional*) – If True, the internal TPSFs are also returned. If False, only boundary TPSFs are returned and data.phi will be empty.

  The default is False.

- **beautify** (*bool, optional*) – If true, zeros the initial unstable parts of the boundary TPSFs. The default is True.

- **solver** (*str, optional*) – Choose between 'CPU' or 'GPU' solver (case insensitive). Automatically determined (GPU prioritized) if not specified

- **opt** (`nirfasterff.utils.SolverOptions`, *optional*) – Solver options. Uses default parameters if not specified, and they should suffice in most cases.

  See `SolverOptions()` for details

Raises

- **TypeError** – If mesh is not a stnd mesh.
- **ValueError** – If tmax is smaller than dt

Returns

- **data** (*nirfasterff.base.TPSFdata*) – internal and boundary TPSFs given the mesh and optodes.

  If mesh.vol is defined and *savefield==True*, the returned internal TPSFs will be in volumetric space

  See `TPSFdata()` for details.

- **info** (*nirfasterff.utils.ConvergenceInfo*) – convergence information of the solver.

  Only the convergence info of the last time step is returned.

  See `ConvergenceInfo()` for details

See also

`gen_mass_matrix()`, `gen_sources()`, and `get_field_TR()`

nirfasterff.forward.femdata.femdata_stnd_TR_moments

nirfasterff.forward.femdata.**femdata_stnd_TR_moments**(*mesh*, *max_moments=3*, *savefield=False*, *solver=utils.get_solver()*, *opt=utils.SolverOptions()*)

Forward modeling calculating TR moments using Mellin transform on a standard mesh. Please consider using mesh.femdata_moments() instead.

The function calculates the MASS matrices, the source vectors, and calls the Mellin moments solver (preconditioned conjugated gradient).

Calculates 0th, 1st, 2nd, .., max_moments-th moments directly without calculating TPSF first.

This is more efficient, if the actual TPSFs are not of concern.

Parameters

- **mesh** (`nirfasterff.base.stndmesh`) – the mesh used to calcuate the forward data.
- **max_moments** (*int32, optional*) – max order of moments to calculate. That is, 0th, 1st, 2nd, .., max_moments-th will be calculated. The default is 3.
- **savefield** (*bool, optional*) – If True, the internal moments are also returned. If False, only boundary moments are returned and data.phi will be empty.

  The default is False.

- **solver** (*str, optional*) – Choose between 'CPU' or 'GPU' solver (case insensitive). Automatically determined (GPU prioritized) if not specified

- **opt** (`nirfasterff.utils.SolverOptions`, *optional*) – Solver options. Uses default parameters if not specified, and they should suffice in most cases.

  See `SolverOptions()` for details

Raises

- **TypeError** – If mesh is not a stnd mesh.
- **ValueError** – If max_moments is negative.

Returns

- **data** (*nirfasterff.base.TRMomentsdata*) – internal and boundary moments given the mesh and optodes.

  If mesh.vol is defined and *savefield==True*, the returned internal moments will be in volumetric space

  See `TRMomentsdata()` for details.

- **info** (*nirfasterff.utils.ConvergenceInfo*) – convergence information of the solver.

  Only the convergence info of highest order moments is returned.

  See `ConvergenceInfo()` for details

See also

`gen_mass_matrix()`, `gen_sources()`, and `get_field_TRmoments()`

References

Arridge and Schweiger, Applied Optics, 1995. doi:10.1364/AO.34.002683

4.1.3 nirfasterff.inverse

Calculation of the Jacobian matrices and a basic Tikhonov regularization function

Functions

`jacobian_DCS`(mesh, tvec[, normalize, ...])	Calculates the Jacobian matrix for a DCS mesh using the adjoint method
`jacobian_fl_CW`(mesh[, normalize, solver, opt])	Calculates the continuous-wave fluorescence Jacobian matrix using the adjoint method
`jacobian_fl_FD`(mesh, freq[, normalize, ...])	Calculates the frequency-domain fluorescence Jacobian matrix using the adjoint method
`jacobian_stnd_CW`(mesh[, normalize, mus, ...])	Calculates the continuous-wave Jacobian matrix using the adjoint method
`jacobian_stnd_FD`(mesh, freq[, normalize, ...])	Calculates the frequency-domain Jacobian matrix using the adjoint method
`tikhonov`(A, reg, y)	Solves Tikhonov regularization (ie ridge regression)

nirfasterff.inverse.jacobian_DCS

nirfasterff.inverse.**jacobian_DCS**(*mesh*, *tvec*, *normalize=True*, *solver=utils.get_solver()*, *opt=utils.SolverOptions()*)

Calculates the Jacobian matrix for a DCS mesh using the adjoint method

One Jacobian is calcualted at each time point in tvec, and the derivative is taken with regard to aDb

Parameters

- **mesh** (*nirfasterff.base.dcsmesh*) – mesh on which the Jacobian is calculated.
- **tvec** (*double NumPy vector*) – time vector used.
- **normalize** (*bool, optional*) – if True, Jacobbians are normalized to the measured boundary amplitude. The default is True.
- **solver** (*str, optional*) – Choose between 'CPU' or 'GPU' solver (case insensitive). Automatically determined (GPU prioritized) if not specified
- **opt** (*nirfasterff.utils.SolverOptions, optional*) – Solver options. Uses default parameters if not specified, and they should suffice in most cases.

  See *SolverOptions()* for details

Raises

TypeError – if mesh is not a DCS mesh, or mesh.vol is not defined.

Returns

- **J** (*double NumPy array*) – The Jacobian matrix. Size (NChannel, NVoxel, NTime)
- **data1** (*nirfasterff.base.FLdata*) – The calculated direct field. The same as directly calling mesh.femdata(tvec)
- **data2** (*nirfasterff.base.FLdata*) – The calculated adjoint field. The same as calling mesh.femdata(tvec) AFTER swapping the locations of sources and detectors

nirfasterff.inverse.jacobian_fl_CW

nirfasterff.inverse.**jacobian_fl_CW**(*mesh*, *normalize=True*, *solver=utils.get_solver()*, *opt=utils.SolverOptions()*)

Calculates the continuous-wave fluorescence Jacobian matrix using the adjoint method

J_{ij} = dA_i / d_gamma_j

where A is fluorescence amplitude if normalization is False, fluorescence amplitude divided by excitation amplitude ('Born ratio') if True

gamma_j = mesh.eta[j]*mesh.muaf[j]

Parameters

- **mesh** (*nirfasterff.base.fluormesh*) – mesh on which the Jacobian is calculated.
- **normalize** (*bool, optional*) – whether normalize the Jacobian to the amplitudes of boundary measurements at excitation wavelength ('Born ratio').

  The default is True.
- **solver** (*str, optional*) – Choose between 'CPU' or 'GPU' solver (case insensitive). Automatically determined (GPU prioritized) if not specified

- **opt** (nirfasterff.utils.SolverOptions, *optional*) – Solver options. Uses default parameters if not specified, and they should suffice in most cases.

  See *SolverOptions()* for details

Raises

TypeError – if mesh is not a fluor mesh, or mesh.vol is not defined.

Returns

- **J** (*double NumPy array*) – The Jacobian matrix. Size (NChannel, NVoxel)
- **data1** (*nirfasterff.base.FLdata*) – The calculated direct field. The same as directly calling mesh.femdata(0)
- **data2** (*nirfasterff.base.FLdata*) – The calculated adjoint field. The same as calling mesh.femdata(0) AFTER swapping the locations of sources and detectors

References

Milstein et al., JOSA A, 2004. doi:10.1364/JOSAA.21.001035

nirfasterff.inverse.jacobian_fl_FD

nirfasterff.inverse.**jacobian_fl_FD**(*mesh*, *freq*, *normalize=True*, *solver=utils.get_solver()*, *opt=utils.SolverOptions()*)

Calculates the frequency-domain fluorescence Jacobian matrix using the adjoint method

The Jacobian is structured as, suppose we have M channels and N voxels:

```
[d_real{A_1}/d_gamma_1, d_real{A_1}/d_gamma_2, ..., d_real{A_1}/d_gamma_{N}, d_real
↪{A_1}/dtau_1, d_real{A_1}/dtau_2, ..., d_real{A_1}/dtau_{N}]
[d_imag{A_1}/d_gamma_1, d_imag{A_1}/d_gamma_2, ..., d_imag{A_1}/d_gamma_{N}, d_imag
↪{A_1}/dtau_1, d_imag{A_1}/dtau_2, ..., d_imag{A_1}/dtau_{N}]
[d_real{A_2}/d_gamma_1, d_real{A_2}/d_gamma_2, ..., d_real{A_2}/d_gamma_{N}, d_real
↪{A_2}/dtau_1, d_real{A_2}/dtau_2, ..., d_real{A_2}/dtau_{N}]
[d_imag{A_2}/d_gamma_1, d_imag{A_2}/d_gamma_2, ..., d_imag{A_2}/d_gamma_{N}, d_imag
↪{A_2}/dtau_1, d_imag{A_2}/dtau_2, ..., d_imag{A_2}/dtau_{N}]
...
[d_real{A_M}/d_gamma_1, d_real{A_M}/d_gamma_2, ..., d_real{A_M}/d_gamma_{N}, d_real
↪{A_M}/dtau_1, d_real{A_M}/dtau_2, ..., d_real{A_M}/dtau_{N}]
[d_imag{A_M}/d_gamma_1, d_imag{A_M}/d_gamma_2, ..., d_imag{A_M}/d_gamma_{N}, d_imag
↪{A_M}/dtau_1, d_imag{A_M}/dtau_2, ..., d_imag{A_M}/dtau_{N}]
```

where A is fluorescence amplitude if normalization is False, fluorescence amplitude divided by excitation amplitude ('Born ratio') if True gamma_j = mesh.eta[j]*mesh.muaf[j]

Parameters

- **mesh** (*nirfasterff.base.fluormesh*) – mesh on which the Jacobian is calculated.
- **freq** (*double*) – modulation frequency in Hz.
- **normalize** (*bool, optional*) – whether normalize the Jacobian to the amplitudes of boundary measurements at excitation wavelength ('Born ratio').

  The default is True.
- **solver** (*str, optional*) – Choose between 'CPU' or 'GPU' solver (case insensitive). Automatically determined (GPU prioritized) if not specified

- **opt** (nirfasterff.utils.SolverOptions, *optional*) – Solver options. Uses default parameters if not specified, and they should suffice in most cases.

  See *SolverOptions()* for details

Raises

TypeError – if mesh is not a fluor mesh, or mesh.vol is not defined.

Returns

- **J** (*double NumPy array*) – The Jacobian matrix. Size (NChannel*2, NVoxel*2)
- **data1** (*nirfasterff.base.FLdata*) – The calculated direct field. The same as directly calling mesh.femdata(freq)
- **data2** (*nirfasterff.base.FLdata*) – The calculated adjoint field. The same as calling mesh.femdata(freq) AFTER swapping the locations of sources and detectors

References

Milstein et al., JOSA A, 2004. doi:10.1364/JOSAA.21.001035

nirfasterff.inverse.jacobian_stnd_CW

nirfasterff.inverse.**jacobian_stnd_CW**(*mesh*, *normalize=True*, *mus=False*, *solver=utils.get_solver()*, *opt=utils.SolverOptions()*)

Calculates the continuous-wave Jacobian matrix using the adjoint method

Calculates spatial distributions of sensitivity of field registerd on the mesh boundary to changes of optical properties per voxel.

When mus is set to True, the Jacobian is structured as, suppose we have M channels and N voxels:

```
[dA_1/dmusp_1, dA_1/dmusp_2, ..., dA_1/dmusp_{N}, dA_1/dmua_1, dA_1/dmua_2, ..., dA_
↪1/dmua_{N}]
[dA_2/dmusp_1, dA_2/dmusp_2, ..., dA_2/dmusp_{N}, dA_2/dmua_1, dA_2/dmua_2, ..., dA_
↪2/dmua_{N}]
...
[dA_M/dmusp_1, dA_M/dmusp_2, ..., dA_M/dmusp_{N}, dA_M/dmua_1, dA_M/dmua_2, ..., dA_
↪M/dmua_{N}]
```

where A and Phi denote the measured amplitude and the phase if *normalize=False*, and the log of them if *normalize=True*

When mus is set to False, the returned Jacobian is only the right half of the above. That is, only derivatives wrt mua

Note that the calculation is only done in the volumetric space

Parameters

- **mesh** (*nirfasterff.base.stndmesh*) – mesh on which the Jacobian is calculated.
- **normalize** (*bool, optional*) – whether normalize the Jacobian to the amplitudes of boundary measurements, i.e. use Rytov approximation.

  The default is True.
- **mus** (*bool, optional*) – whether derivates wrt mus (left half of the 'full' Jacobian) is calculated. The default is False.
- **solver** (*str, optional*) – Choose between 'CPU' or 'GPU' solver (case insensitive). Automatically determined (GPU prioritized) if not specified

- **opt** (nirfasterff.utils.SolverOptions, *optional*) – Solver options. Uses default parameters if not specified, and they should suffice in most cases.

  See *SolverOptions()* for details

Raises

TypeError – if mesh is not a stnd mesh, or mesh.vol is not defined.

Returns

- **J** (*double NumPy array*) – The Jacobian matrix. Size (NChannel, NVoxel*2) if mus=True, (NChannel, NVoxel) if mus=False
- **data1** (*nirfasterff.base.FDdata*) – The calculated direct field. The same as directly calling mesh.femdata(0)
- **data2** (*nirfasterff.base.FDdata*) – The calculated adjoint field. The same as calling mesh.femdata(0) AFTER swapping the locations of sources and detectors

nirfasterff.inverse.jacobian_stnd_FD

nirfasterff.inverse.**jacobian_stnd_FD**(*mesh*, *freq*, *normalize=True*, *mus=True*, *solver=utils.get_solver()*, *opt=utils.SolverOptions()*)

Calculates the frequency-domain Jacobian matrix using the adjoint method

Calculates spatial distributions of sensitivity of field registerd on the mesh boundary to changes of optical properties per voxel.

When freq=0, see *jacobian_stnd_CW()* for the structure of the Jacobian

When freq>0, the Jacobian is structured as, suppose we have M channels and N voxels:

```
[dA_1/dmusp_1, dA_1/dmusp_2, ..., dA_1/dmusp_{N}, dA_1/dmua_1, dA_1/dmua_2, ..., dA_
↪1/dmua_{N}]
[dPhi_1/dmusp_1, dPhi_1/dmusp_2, ..., dPhi_1/dmusp_{N}, dPhi_1/dmua_1, dPhi_1/dmua_
↪2, ..., dPhi_1/dmua_{N}]
[dA_2/dmusp_1, dA_2/dmusp_2, ..., dA_2/dmusp_{N}, dA_2/dmua_1, dA_2/dmua_2, ..., dA_
↪2/dmua_{N}]
[dPhi_2/dmusp_1, dPhi_2/dmusp_2, ..., dPhi_2/dmusp_{N}, dPhi_2/dmua_1, dPhi_2/dmua_
↪2, ..., dPhi_2/dmua_{N}]
...
[dA_M/dmusp_1, dA_M/dmusp_2, ..., dA_M/dmusp_{N}, dA_M/dmua_1, dA_M/dmua_2, ..., dA_
↪M/dmua_{N}]
[dPhi_M/dmusp_1, dPhi_M/dmusp_2, ..., dPhi_M/dmusp_{N}, dPhi_M/dmua_1, dPhi_M/dmua_
↪2, ..., dPhi_M/dmua_{N}]
```

where A and Phi denote the measured amplitude and the phase if *normalize=False*, and the log of them if *normalize=True*

Note that the calculation is only done in the volumetric space

Parameters

- **mesh** (*nirfasterff.base.stndmesh*) – mesh on which the Jacobian is calculated.
- **freq** (*double*) – modulation frequency, in Hz.
- **normalize** (*bool, optional*) – whether normalize the Jacobian to the amplitudes of boundary measurements, i.e. use Rytov approximation.

  The default is True.

- **mus** (*bool, optional*) – whether derivates wrt mus (left half of the 'full' Jacobian) is calculated. Only has effect when freq=0. The default is True.
- **solver** (*str, optional*) – Choose between 'CPU' or 'GPU' solver (case insensitive). Automatically determined (GPU prioritized) if not specified
- **opt** (`nirfasterff.utils.SolverOptions`, *optional*) – Solver options. Uses default parameters if not specified, and they should suffice in most cases.

  See *SolverOptions()* for details

Raises

TypeError – if mesh is not a stnd mesh, or mesh.vol is not defined.

Returns

- **J** (*double or complex double NumPy array*) – The Jacobian matrix. Size (NChannel*2, NVoxel*2) if freq>0, (NChannel, NVoxel*2) if freq=0 and mus=True, (NChannel, NVoxel) if freq=0 and mus=False
- **data1** (*nirfasterff.base.FDdata*) – The calculated direct field. The same as directly calling mesh.femdata(freq)
- **data2** (*nirfasterff.base.FDdata*) – The calculated adjoint field. The same as calling mesh.femdata(freq) AFTER swapping the locations of sources and detectors

References

Arridge, Applied Optics, 1995. doi:10.1364/AO.34.007395

nirfasterff.inverse.tikhonov

`nirfasterff.inverse.`**`tikhonov`**`(`*A*, *reg*, *y*`)`

Solves Tikhonov regularization (ie ridge regression)

That is, given a linear system y = Ax, it solves $\arg\min_x ||Ax - y||_2^2 + ||\Gamma x||_2^2$

where A is the forward matrix, y is the recording, and Γ is the regularization matrix

Parameters

- **A** (*double NumPy array*) – forward matrix, e.g. the Jacobian in DOT.
- **reg** (*double NumPy array or scalar*) – if a scalar, the same regularization is applied to all elements of x, i.e. $\Gamma = reg * I$.

  if a vector, the regularization matrix is assumed to be diagonal, with the diagonal elements specified in reg, i.e. $\Gamma = diag(reg)$.

  if a matrix, it must be symmetric. In this case, $\Gamma = reg$.
- **y** (*double NumPy vector*) – measurement vector, e.g. dOD at each channel in DOT.

Raises

ValueError – if reg is of incompatible size or not symmetric (if matrix).

Returns

result – Tikhonov regularized solution to the linear system.

Return type

double NumPy vector

4.1.4 nirfasterff.io

Some functions for reading/writing certain data types.

As of now, they are only used by the CGAL mesher, and there should be no need for the user to directly call them.

Functions

readMEDIT(fname)	Read a mesh generated by the CGAL mesher, which is saved in MEDIT format
saveinr(vol, fname[, xPixelSpacing, ...])	Save a volume in the INRIA format.

nirfasterff.io.readMEDIT

`nirfasterff.io.`**`readMEDIT`**`(`*fname*`)`

Read a mesh generated by the CGAL mesher, which is saved in MEDIT format

Directly translated from the Matlab version

Parameters
fname (*str*) – name of the file to be loaded.

Returns

- **elements** (*NumPy array*) – list of elements in the mesh. Zero-based
- **nodes** (*NumPy array*) – node locations of the mesh, in mm.
- **faces** (*NumPy array*) – list of faces in the mesh. In case of 2D, it's the same as elements. Zero-based
- **nnpe** (*int*) – size of dimension 1 of elements, i.e. 4 for 3D mesh and 3 for 2D mesh.

nirfasterff.io.saveinr

`nirfasterff.io.`**`saveinr`**`(`*vol*, *fname*, *xPixelSpacing=1.*, *yPixelSpacing=1.*, *SliceThickness=1.*`)`

Save a volume in the INRIA format. This is for the CGAL mesher.

Directly translated from the Matlab version

Parameters

- **vol** (*NumPy array*) – the volume to be saved.
- **fname** (*str*) – file name to be saved as.
- **xPixelSpacing** (*double, optional*) – volume resolution in x direction. The default is 1..
- **yPixelSpacing** (*double, optional*) – volume resolution in y direction. The default is 1..
- **SliceThickness** (*double, optional*) – volume resolution in z direction. The default is 1..

Return type
None.

4.1.5 nirfasterff.lib

Low-level functions implemented in C/C++, on both GPU and CPU

You should NOT directly call these functions. Please use the wrapper functions provided instead

Modules

nirfasterff_cpu
nirfasterff_cuda

nirfasterff.lib.nirfasterff_cpu

Functions

IntGradGrid(*args, **kwargs)	Overloaded function.
IntGrid(*args, **kwargs)	Overloaded function.
ele_area(arg0, arg1)	
gen_mass_matrix(arg0, arg1, arg2, arg3, ...)	
gen_source_fl(*args, **kwargs)	Overloaded function.
get_field_CW(*args, **kwargs)	Overloaded function.
get_field_FD(*args, **kwargs)	Overloaded function.
get_field_TR(*args, **kwargs)	Overloaded function.
get_field_TRFL(arg0, arg1, arg2, arg3, arg4, ...)	
get_field_TRFL_moments(arg0, arg1, arg2, ...)	
get_field_TR_moments(*args, **kwargs)	Overloaded function.
gradientIntfunc(arg0, arg1, arg2)	
gradientIntfunc2(arg0, arg1, arg2)	
isCUDA()	
mesh_support(arg0, arg1, arg2)	
pointLocation(arg0, arg1, arg2)	

nirfasterff.lib.nirfasterff_cpu.IntGradGrid

`nirfasterff.lib.nirfasterff_cpu.`**`IntGradGrid`**(**args*, ***kwargs*)

Overloaded function.

1. IntGradGrid(arg0: numpy.ndarray[numpy.complex128[m, n], flags.writeable, flags.c_contiguous], arg1: numpy.ndarray[numpy.complex128[m, n], flags.writeable, flags.c_contiguous], arg2: numpy.ndarray[numpy.complex128[m, n], flags.writeable, flags.c_contiguous], arg3: numpy.ndarray[numpy.complex128[m, n], flags.writeable, flags.c_contiguous], arg4: numpy.ndarray[numpy.complex128[m, n], flags.writeable, flags.c_contiguous],

arg5: numpy.ndarray[numpy.complex128[m, n], flags.writeable, flags.c_contiguous], arg6: numpy.ndarray[numpy.int32[m, n], flags.writeable, flags.c_contiguous], arg7: int) -> numpy.ndarray[numpy.complex128[m, n]]

2. IntGradGrid(arg0: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous], arg1: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous], arg2: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous], arg3: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous], arg4: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous], arg5: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous], arg6: numpy.ndarray[numpy.int32[m, n], flags.writeable, flags.c_contiguous], arg7: int) -> numpy.ndarray[numpy.float64[m, n]]

nirfasterff.lib.nirfasterff_cpu.IntGrid

`nirfasterff.lib.nirfasterff_cpu.`**`IntGrid`**`(`**args*, ***kwargs*`)`

Overloaded function.

1. IntGrid(arg0: numpy.ndarray[numpy.complex128[m, n], flags.writeable, flags.c_contiguous], arg1: numpy.ndarray[numpy.complex128[m, n], flags.writeable, flags.c_contiguous], arg2: numpy.ndarray[numpy.int32[m, n], flags.writeable, flags.c_contiguous]) -> numpy.ndarray[numpy.complex128[m, n]]

2. IntGrid(arg0: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous], arg1: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous], arg2: numpy.ndarray[numpy.int32[m, n], flags.writeable, flags.c_contiguous]) -> numpy.ndarray[numpy.float64[m, n]]

nirfasterff.lib.nirfasterff_cpu.ele_area

`nirfasterff.lib.nirfasterff_cpu.`**`ele_area`**`(`*arg0: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous]*, *arg1: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous]*`)` → numpy.ndarray[numpy.float64[m, 1]]

nirfasterff.lib.nirfasterff_cpu.gen_mass_matrix

`nirfasterff.lib.nirfasterff_cpu.`**`gen_mass_matrix`**`(`*arg0: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous]*, *arg1: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous]*, *arg2: numpy.ndarray[numpy.float64[m, 1], flags.writeable]*, *arg3: numpy.ndarray[numpy.float64[m, 1], flags.writeable]*, *arg4: numpy.ndarray[numpy.float64[m, 1], flags.writeable]*, *arg5: numpy.ndarray[numpy.float64[m, 1], flags.writeable]*, *arg6: numpy.ndarray[numpy.float64[m, 1], flags.writeable]*, *arg7: float*`)` → tuple[numpy.ndarray[numpy.int32[m, 1]], numpy.ndarray[numpy.int32[m, 1]], numpy.ndarray[numpy.complex128[m, 1]]]

nirfasterff.lib.nirfasterff_cpu.gen_source_fl

`nirfasterff.lib.nirfasterff_cpu.`**`gen_source_fl`**`(`**args*, ***kwargs*`)`

Overloaded function.

1. gen_source_fl(arg0: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous], arg1: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous], arg2: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous]) -> numpy.ndarray[numpy.float64[m, n]]
2. gen_source_fl(arg0: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous], arg1: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous], arg2: numpy.ndarray[numpy.complex128[m, n], flags.writeable, flags.c_contiguous]) -> numpy.ndarray[numpy.complex128[m, n]]

nirfasterff.lib.nirfasterff_cpu.get_field_CW

`nirfasterff.lib.nirfasterff_cpu.`**`get_field_CW`**`(`**args*, ***kwargs*`)`

Overloaded function.

1. get_field_CW(arg0: numpy.ndarray[numpy.int32[m, 1], flags.writeable], arg1: numpy.ndarray[numpy.int32[m, 1], flags.writeable], arg2: numpy.ndarray[numpy.float64[m, 1], flags.writeable], arg3: scipy.sparse.csc_matrix[numpy.float64], max_iter: int = 1000, AbsoluteTolerance: float = 1e-12, RelativeTolerance: float = 1e-12, divergence: float = 100000000.0, nthread: int = 8) -> tuple[numpy.ndarray[numpy.float64[m, n]], list[nirfasterff.lib.nirfasterff_cpu.ConvergenceInfoCPU]]
2. get_field_CW(arg0: numpy.ndarray[numpy.int32[m, 1], flags.writeable], arg1: numpy.ndarray[numpy.int32[m, 1], flags.writeable], arg2: numpy.ndarray[numpy.float64[m, 1], flags.writeable], arg3: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous], max_iter: int = 1000, AbsoluteTolerance: float = 1e-12, RelativeTolerance: float = 1e-12, divergence: float = 100000000.0, nthread: int = 8) -> tuple[numpy.ndarray[numpy.float64[m, n]], list[nirfasterff.lib.nirfasterff_cpu.ConvergenceInfoCPU]]

nirfasterff.lib.nirfasterff_cpu.get_field_FD

`nirfasterff.lib.nirfasterff_cpu.`**`get_field_FD`**`(`**args*, ***kwargs*`)`

Overloaded function.

1. get_field_FD(arg0: numpy.ndarray[numpy.int32[m, 1], flags.writeable], arg1: numpy.ndarray[numpy.int32[m, 1], flags.writeable], arg2: numpy.ndarray[numpy.complex128[m, 1], flags.writeable], arg3: scipy.sparse.csc_matrix[numpy.complex128], max_iter: int = 1000, AbsoluteTolerance: float = 1e-12, RelativeTolerance: float = 1e-12, divergence: float = 100000000.0, nthread: int = 8) -> tuple[numpy.ndarray[numpy.complex128[m, n]], list[nirfasterff.lib.nirfasterff_cpu.ConvergenceInfoCPU]]
2. get_field_FD(arg0: numpy.ndarray[numpy.int32[m, 1], flags.writeable], arg1: numpy.ndarray[numpy.int32[m, 1], flags.writeable], arg2: numpy.ndarray[numpy.complex128[m, 1], flags.writeable], arg3: numpy.ndarray[numpy.complex128[m, n], flags.writeable, flags.c_contiguous], max_iter: int = 1000, AbsoluteTolerance: float = 1e-12, RelativeTolerance: float = 1e-12, divergence: float = 100000000.0, nthread: int = 8) -> tuple[numpy.ndarray[numpy.complex128[m, n]], list[nirfasterff.lib.nirfasterff_cpu.ConvergenceInfoCPU]]

nirfasterff.lib.nirfasterff_cpu.get_field_TR

`nirfasterff.lib.nirfasterff_cpu.`**`get_field_TR`**`(`**args*, ***kwargs*`)`

Overloaded function.

1. get_field_TR(arg0: numpy.ndarray[numpy.int32[m, 1], flags.writeable], arg1: numpy.ndarray[numpy.int32[m, 1], flags.writeable], arg2: numpy.ndarray[numpy.float64[m, 1], flags.writeable], arg3: numpy.ndarray[numpy.float64[m, 1], flags.writeable], arg4: scipy.sparse.csc_matrix[numpy.float64], arg5: int, max_iter: int = 1000, AbsoluteTolerance: float = 1e-12, RelativeTolerance: float = 1e-12, divergence: float = 100000000.0, nthread: int = 8) -> tuple[numpy.ndarray[numpy.float64[m, n]], list[nirfasterff.lib.nirfasterff_cpu.ConvergenceInfoCPU]]
2. get_field_TR(arg0: numpy.ndarray[numpy.int32[m, 1], flags.writeable], arg1: numpy.ndarray[numpy.int32[m, 1], flags.writeable], arg2: numpy.ndarray[numpy.float64[m, 1], flags.writeable], arg3: numpy.ndarray[numpy.float64[m, 1], flags.writeable], arg4: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous], arg5: int, max_iter: int = 1000, AbsoluteTolerance: float = 1e-12, RelativeTolerance: float = 1e-12, divergence: float = 100000000.0, nthread: int = 8) -> tuple[numpy.ndarray[numpy.float64[m, n]], list[nirfasterff.lib.nirfasterff_cpu.ConvergenceInfoCPU]]

nirfasterff.lib.nirfasterff_cpu.get_field_TRFL

`nirfasterff.lib.nirfasterff_cpu.`**`get_field_TRFL`**(*arg0: numpy.ndarray[numpy.int32[m, 1], flags.writeable]*, *arg1: numpy.ndarray[numpy.int32[m, 1], flags.writeable]*, *arg2: numpy.ndarray[numpy.float64[m, 1], flags.writeable]*, *arg3: numpy.ndarray[numpy.float64[m, 1], flags.writeable]*, *arg4: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous]*, *arg5: int*, *max_iter: int = 1000*, *AbsoluteTolerance: float = 1e-12*, *RelativeTolerance: float = 1e-12*, *divergence: float = 100000000.0*, *nthread: int = 8*) → tuple[numpy.ndarray[numpy.float64[m, n]], list[*nirfasterff.lib.nirfasterff_cpu.ConvergenceInfoCPU*]]

nirfasterff.lib.nirfasterff_cpu.get_field_TRFL_moments

`nirfasterff.lib.nirfasterff_cpu.`**`get_field_TRFL_moments`**(*arg0: numpy.ndarray[numpy.int32[m, 1], flags.writeable]*, *arg1: numpy.ndarray[numpy.int32[m, 1], flags.writeable]*, *arg2: numpy.ndarray[numpy.float64[m, 1], flags.writeable]*, *arg3: numpy.ndarray[numpy.float64[m, 1], flags.writeable]*, *arg4: numpy.ndarray[numpy.float64[m, 1], flags.writeable]*, *arg5: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous]*, *arg6: numpy.ndarray[numpy.float64[m, 1], flags.writeable]*, *arg7: numpy.ndarray[numpy.float64[m, 1], flags.writeable]*, *arg8: int*, *max_iter: int = 1000*, *AbsoluteTolerance: float = 1e-12*, *RelativeTolerance: float = 1e-12*, *divergence: float = 100000000.0*, *nthread: int = 8*) → tuple[numpy.ndarray[numpy.float64[m, n]], list[*nirfasterff.lib.nirfasterff_cpu.ConvergenceInfoCPU*]]

nirfasterff.lib.nirfasterff_cpu.get_field_TR_moments

`nirfasterff.lib.nirfasterff_cpu.`**`get_field_TR_moments`**`(`**args*, ***kwargs*`)`

Overloaded function.

1. get_field_TR_moments(arg0: numpy.ndarray[numpy.int32[m, 1], flags.writeable], arg1: numpy.ndarray[numpy.int32[m, 1], flags.writeable], arg2: numpy.ndarray[numpy.float64[m, 1], flags.writeable], arg3: numpy.ndarray[numpy.float64[m, 1], flags.writeable], arg4: scipy.sparse.csc_matrix[numpy.float64], arg5: int, max_iter: int = 1000, AbsoluteTolerance: float = 1e-12, RelativeTolerance: float = 1e-12, divergence: float = 100000000.0, nthread: int = 8) -> tuple[numpy.ndarray[numpy.float64[m, n]], list[nirfasterff.lib.nirfasterff_cpu.ConvergenceInfoCPU]]
2. get_field_TR_moments(arg0: numpy.ndarray[numpy.int32[m, 1], flags.writeable], arg1: numpy.ndarray[numpy.int32[m, 1], flags.writeable], arg2: numpy.ndarray[numpy.float64[m, 1], flags.writeable], arg3: numpy.ndarray[numpy.float64[m, 1], flags.writeable], arg4: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous], arg5: int, max_iter: int = 1000, AbsoluteTolerance: float = 1e-12, RelativeTolerance: float = 1e-12, divergence: float = 100000000.0, nthread: int = 8) -> tuple[numpy.ndarray[numpy.float64[m, n]], list[nirfasterff.lib.nirfasterff_cpu.ConvergenceInfoCPU]]

nirfasterff.lib.nirfasterff_cpu.gradientIntfunc

`nirfasterff.lib.nirfasterff_cpu.`**`gradientIntfunc`**`(`*arg0: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous]*, *arg1: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous]*, *arg2: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous]*`)` → tuple[numpy.ndarray[numpy.int32[m, 1]], numpy.ndarray[numpy.float64[m, n]]]

nirfasterff.lib.nirfasterff_cpu.gradientIntfunc2

`nirfasterff.lib.nirfasterff_cpu.`**`gradientIntfunc2`**`(`*arg0: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous]*, *arg1: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous]*, *arg2: numpy.ndarray[numpy.int32[m, 1], flags.writeable]*`)` → numpy.ndarray[numpy.float64[m, n]]

nirfasterff.lib.nirfasterff_cpu.isCUDA

`nirfasterff.lib.nirfasterff_cpu.`**`isCUDA`**`()` → bool

nirfasterff.lib.nirfasterff_cpu.mesh_support

`nirfasterff.lib.nirfasterff_cpu.`**`mesh_support`**`(`*arg0: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous]*, *arg1: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous]*, *arg2: numpy.ndarray[numpy.float64[m, 1], flags.writeable]*`)` → numpy.ndarray[numpy.float64[m, 1]]

nirfasterff.lib.nirfasterff_cpu.pointLocation

`nirfasterff.lib.nirfasterff_cpu.`**`pointLocation`**(*arg0: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous]*, *arg1: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous]*, *arg2: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous]*) → tuple[numpy.ndarray[numpy.int32[m, 1]], numpy.ndarray[numpy.float64[m, n]]]

Classes

`ConvergenceInfoCPU`

nirfasterff.lib.nirfasterff_cpu.ConvergenceInfoCPU

class `nirfasterff.lib.nirfasterff_cpu.`**`ConvergenceInfoCPU`**

Bases: `pybind11_object`

`__init__`(*self: nirfasterff.lib.nirfasterff_cpu.ConvergenceInfoCPU*) → None

Methods

`__init__`(self)

Attributes

`isConverged`
`isConvergedToAbsoluteTolerance`
`iteration`
`residual`

nirfasterff.lib.nirfasterff_cuda

Functions

gen_mass_matrix(arg0, arg1, arg2, arg3, ...)	
gen_source_fl(*args, **kwargs)	Overloaded function.
get_field_CW(*args, **kwargs)	Overloaded function.
get_field_FD(*args, **kwargs)	Overloaded function.
get_field_TR(*args, **kwargs)	Overloaded function.
get_field_TRFL(arg0, arg1, arg2, arg3, arg4, ...)	
get_field_TRFL_moments(arg0, arg1, arg2, ...)	
get_field_TR_moments(*args, **kwargs)	Overloaded function.

nirfasterff.lib.nirfasterff_cuda.gen_mass_matrix

`nirfasterff.lib.nirfasterff_cuda.`**`gen_mass_matrix`**(*arg0: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous]*, *arg1: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous]*, *arg2: numpy.ndarray[numpy.float64[m, 1], flags.writeable]*, *arg3: numpy.ndarray[numpy.float64[m, 1], flags.writeable]*, *arg4: numpy.ndarray[numpy.float64[m, 1], flags.writeable]*, *arg5: numpy.ndarray[numpy.float64[m, 1], flags.writeable]*, *arg6: numpy.ndarray[numpy.float64[m, 1], flags.writeable]*, *arg7: float*, *GPU: int = -1*) → tuple[numpy.ndarray[numpy.int32[m, 1]], numpy.ndarray[numpy.int32[m, 1]], numpy.ndarray[numpy.complex128[m, 1]]]

nirfasterff.lib.nirfasterff_cuda.gen_source_fl

`nirfasterff.lib.nirfasterff_cuda.`**`gen_source_fl`**(**args*, ***kwargs*)

Overloaded function.

1. gen_source_fl(arg0: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous], arg1: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous], arg2: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous], GPU: int = -1) -> numpy.ndarray[numpy.float64[m, n]]

2. gen_source_fl(arg0: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous], arg1: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous], arg2: numpy.ndarray[numpy.complex128[m, n], flags.writeable, flags.c_contiguous], GPU: int = -1) -> numpy.ndarray[numpy.complex128[m, n]]

nirfasterff.lib.nirfasterff_cuda.get_field_CW

`nirfasterff.lib.nirfasterff_cuda.`**`get_field_CW`**`(`*`*args`*`, `*`**kwargs`*`)`

Overloaded function.

1. get_field_CW(arg0: numpy.ndarray[numpy.int32[m, 1], flags.writeable], arg1: numpy.ndarray[numpy.int32[m, 1], flags.writeable], arg2: numpy.ndarray[numpy.float64[m, 1], flags.writeable], arg3: scipy.sparse.csc_matrix[numpy.float64], max_iter: int = 1000, AbsoluteTolerance: float = 1e-12, RelativeTolerance: float = 1e-12, divergence: float = 100000000.0, GPU: int = -1) -> tuple[numpy.ndarray[numpy.float64[m, n]], list[nirfasterff.lib.nirfasterff_cuda.ConvergenceInfoGPU]]
2. get_field_CW(arg0: numpy.ndarray[numpy.int32[m, 1], flags.writeable], arg1: numpy.ndarray[numpy.int32[m, 1], flags.writeable], arg2: numpy.ndarray[numpy.float64[m, 1], flags.writeable], arg3: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous], max_iter: int = 1000, AbsoluteTolerance: float = 1e-12, RelativeTolerance: float = 1e-12, divergence: float = 100000000.0, GPU: int = -1) -> tuple[numpy.ndarray[numpy.float64[m, n]], list[nirfasterff.lib.nirfasterff_cuda.ConvergenceInfoGPU]]

nirfasterff.lib.nirfasterff_cuda.get_field_FD

`nirfasterff.lib.nirfasterff_cuda.`**`get_field_FD`**`(`*`*args`*`, `*`**kwargs`*`)`

Overloaded function.

1. get_field_FD(arg0: numpy.ndarray[numpy.int32[m, 1], flags.writeable], arg1: numpy.ndarray[numpy.int32[m, 1], flags.writeable], arg2: numpy.ndarray[numpy.complex128[m, 1], flags.writeable], arg3: scipy.sparse.csc_matrix[numpy.complex128], max_iter: int = 1000, AbsoluteTolerance: float = 1e-12, RelativeTolerance: float = 1e-12, divergence: float = 100000000.0, GPU: int = -1) -> tuple[numpy.ndarray[numpy.complex128[m, n]], list[nirfasterff.lib.nirfasterff_cuda.ConvergenceInfoGPU]]
2. get_field_FD(arg0: numpy.ndarray[numpy.int32[m, 1], flags.writeable], arg1: numpy.ndarray[numpy.int32[m, 1], flags.writeable], arg2: numpy.ndarray[numpy.complex128[m, 1], flags.writeable], arg3: numpy.ndarray[numpy.complex128[m, n], flags.writeable, flags.c_contiguous], max_iter: int = 1000, AbsoluteTolerance: float = 1e-12, RelativeTolerance: float = 1e-12, divergence: float = 100000000.0, GPU: int = -1) -> tuple[numpy.ndarray[numpy.complex128[m, n]], list[nirfasterff.lib.nirfasterff_cuda.ConvergenceInfoGPU]]

nirfasterff.lib.nirfasterff_cuda.get_field_TR

`nirfasterff.lib.nirfasterff_cuda.`**`get_field_TR`**`(`*`*args`*`, `*`**kwargs`*`)`

Overloaded function.

1. get_field_TR(arg0: numpy.ndarray[numpy.int32[m, 1], flags.writeable], arg1: numpy.ndarray[numpy.int32[m, 1], flags.writeable], arg2: numpy.ndarray[numpy.float64[m, 1], flags.writeable], arg3: numpy.ndarray[numpy.float64[m, 1], flags.writeable], arg4: scipy.sparse.csc_matrix[numpy.float64], arg5: int, max_iter: int = 1000, AbsoluteTolerance: float = 1e-12, RelativeTolerance: float = 1e-12, divergence: float = 100000000.0, GPU: int = -1) -> tuple[numpy.ndarray[numpy.float64[m, n]], list[nirfasterff.lib.nirfasterff_cuda.ConvergenceInfoGPU]]
2. get_field_TR(arg0: numpy.ndarray[numpy.int32[m, 1], flags.writeable], arg1: numpy.ndarray[numpy.int32[m, 1], flags.writeable], arg2: numpy.ndarray[numpy.float64[m, 1], flags.writeable], arg3: numpy.ndarray[numpy.float64[m, 1], flags.writeable], arg4: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous], arg5: int, max_iter: int = 1000, AbsoluteTolerance: float = 1e-12, RelativeTolerance: float = 1e-12, divergence: float = 100000000.0, GPU: int = -1) -> tuple[numpy.ndarray[numpy.float64[m, n]], list[nirfasterff.lib.nirfasterff_cuda.ConvergenceInfoGPU]]

nirfasterff.lib.nirfasterff_cuda.get_field_TRFL

`nirfasterff.lib.nirfasterff_cuda.`**`get_field_TRFL`**(*arg0: numpy.ndarray[numpy.int32[m, 1], flags.writeable]*, *arg1: numpy.ndarray[numpy.int32[m, 1], flags.writeable]*, *arg2: numpy.ndarray[numpy.float64[m, 1], flags.writeable]*, *arg3: numpy.ndarray[numpy.float64[m, 1], flags.writeable]*, *arg4: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous]*, *arg5: int*, *max_iter: int = 1000*, *AbsoluteTolerance: float = 1e-12*, *RelativeTolerance: float = 1e-12*, *divergence: float = 100000000.0*, *GPU: int = -1*) → tuple[numpy.ndarray[numpy.float64[m, n]], list[*nirfasterff.lib.nirfasterff_cuda.ConvergenceInfoGPU*]]

nirfasterff.lib.nirfasterff_cuda.get_field_TRFL_moments

`nirfasterff.lib.nirfasterff_cuda.`**`get_field_TRFL_moments`**(*arg0: numpy.ndarray[numpy.int32[m, 1], flags.writeable]*, *arg1: numpy.ndarray[numpy.int32[m, 1], flags.writeable]*, *arg2: numpy.ndarray[numpy.float64[m, 1], flags.writeable]*, *arg3: numpy.ndarray[numpy.float64[m, 1], flags.writeable]*, *arg4: numpy.ndarray[numpy.float64[m, 1], flags.writeable]*, *arg5: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous]*, *arg6: numpy.ndarray[numpy.float64[m, 1], flags.writeable]*, *arg7: numpy.ndarray[numpy.float64[m, 1], flags.writeable]*, *arg8: int*, *max_iter: int = 1000*, *AbsoluteTolerance: float = 1e-12*, *RelativeTolerance: float = 1e-12*, *divergence: float = 100000000.0*, *GPU: int = -1*) → tuple[numpy.ndarray[numpy.float64[m, n]], list[*nirfasterff.lib.nirfasterff_cuda.ConvergenceInfoGPU*]]

nirfasterff.lib.nirfasterff_cuda.get_field_TR_moments

`nirfasterff.lib.nirfasterff_cuda.`**`get_field_TR_moments`**(**args*, ***kwargs*)

Overloaded function.

1. get_field_TR_moments(arg0: numpy.ndarray[numpy.int32[m, 1], flags.writeable], arg1: numpy.ndarray[numpy.int32[m, 1], flags.writeable], arg2: numpy.ndarray[numpy.float64[m, 1], flags.writeable], arg3: numpy.ndarray[numpy.float64[m, 1], flags.writeable], arg4: scipy.sparse.csc_matrix[numpy.float64], arg5: int, max_iter: int = 1000, AbsoluteTolerance: float = 1e-12, RelativeTolerance: float = 1e-12, divergence: float = 100000000.0, GPU: int = -1) -> tuple[numpy.ndarray[numpy.float64[m, n]], list[nirfasterff.lib.nirfasterff_cuda.ConvergenceInfoGPU]]
2. get_field_TR_moments(arg0: numpy.ndarray[numpy.int32[m, 1], flags.writeable], arg1: numpy.ndarray[numpy.int32[m, 1], flags.writeable], arg2: numpy.ndarray[numpy.float64[m,

1], flags.writeable], arg3: numpy.ndarray[numpy.float64[m, 1], flags.writeable], arg4: numpy.ndarray[numpy.float64[m, n], flags.writeable, flags.c_contiguous], arg5: int, max_iter: int = 1000, AbsoluteTolerance: float = 1e-12, RelativeTolerance: float = 1e-12, divergence: float = 100000000.0, GPU: int = -1) -> tuple[numpy.ndarray[numpy.float64[m, n]], list[nirfasterff.lib.nirfasterff_cuda.ConvergenceInfoGPU]]

Classes

ConvergenceInfoGPU

nirfasterff.lib.nirfasterff_cuda.ConvergenceInfoGPU

`class nirfasterff.lib.nirfasterff_cuda.`**`ConvergenceInfoGPU`**

Bases: `pybind11_object`

`__init__`(*self*: nirfasterff.lib.nirfasterff_cuda.ConvergenceInfoGPU) → None

Methods

__init__(self)

Attributes

`isConverged`
`isConvergedToAbsoluteTolerance`
`iteration`
`residual`

4.1.6 nirfasterff.math

Some low-level functions used by the forward solvers.

It is usually unnecessary to use them directly and caution must be excercised, as many of them interact closely with the C++ libraries and can cause crashes if used incorrectly

Functions

gen_mass_matrix(mesh, omega[, solver, GPU])	Calculate the MASS matrix, and return the coordinates in CSR format.
gen_sources(mesh)	Calculate the source vectors (point source only) for the sources in mesh.source field
gen_sources_fl(mesh, phix[, frequency, ...])	Calculates FEM sources vector for re-emission.
get_boundary_data(mesh, phi)	Calculates boundary data given the field data in mesh
get_field_CW(csrI, csrJ, csrV, qvec[, opt, ...])	Call the Preconditioned Conjugate Gradient solver with FSAI preconditioner.
get_field_FD(csrI, csrJ, csrV, qvec[, opt, ...])	Call the Preconditioned BiConjugate Stablized solver with FSAI preconditioner.
get_field_TR(csrI, csrJ, csrV, qvec, dt, ...)	Call the Preconditioned Conjugate Gradient solver with FSAI preconditioner.
get_field_TRFL(csrI, csrJ, csrV, qvec_m, dt, ...)	Call the Preconditioned Conjugate Gradient solver with FSAI preconditioner.
get_field_TRFLmoments(csrI, csrJ, csrV, ...)	Call the Preconditioned Conjugate Gradient solver with FSAI preconditioner.
get_field_TRmoments(csrI, csrJ, csrV, qvec, ...)	Call the Preconditioned Conjugate Gradient solver with FSAI preconditioner.

nirfasterff.math.gen_mass_matrix

nirfasterff.math.**gen_mass_matrix**(*mesh*, *omega*, *solver=utils.get_solver()*, *GPU=-1*)

Calculate the MASS matrix, and return the coordinates in CSR format.

The current Matlab version outputs COO format, so the results are NOT directly compatible

If calculation fails on GPU (if chosen), it will generate a warning and automatically switch to CPU

Parameters

- **mesh** (*nirfasterff.base.stndmesh*) – the mesh used to calculate the MASS matrix.
- **omega** (*double*) – modulation frequency, in radian.
- **solver** (*str, optional*) – Choose between 'CPU' or 'GPU' solver (case insensitive). Automatically determined (GPU prioritized) if not specified
- **GPU** (*int, optional*) – GPU selection. -1 for automatic, 0, 1, … for manual selection on multi-GPU systems. The default is -1.

Raises

- **RuntimeError** – if both CUDA and CPU versions fail.
- **TypeError** – if 'solver' is not 'CPU' or 'GPU'.

Returns

- **csrI** (*int32 NumPy vector, zero-based*) – I indices of the MASS matrix, in CSR format. Size (NNodes,)
- **csrJ** (*int32 NumPy vector, zero-based*) – J indices of the MASS matrix, in CSR format. Size (nnz(MASS),)
- **csrV** (*float64 or complex128 NumPy vector*) – values of the MASS matrix, in CSR format. Size (nnz(MASS),)

nirfasterff.math.gen_sources

`nirfasterff.math.`**`gen_sources`**(*mesh*)

Calculate the source vectors (point source only) for the sources in mesh.source field

Parameters
`mesh` (*`NIRFASTer mesh type`*) – mesh used to calculate the source vectors. Source information is also defined here.

Returns
qvec – source vectors, where each column corresponds to one source. Size (NNodes, Nsources).

Return type
complex double NumPy array

nirfasterff.math.gen_sources_fl

`nirfasterff.math.`**`gen_sources_fl`**(*mesh*, *phix*, *frequency=0.*, *solver=utils.get_solver()*, *GPU=-1*)

Calculates FEM sources vector for re-emission.

If calculation fails on GPU (if chosen), it will generate a warning and automatically switch to CPU.

Parameters

- **`mesh`** (*`nirfasterff.base.fluormesh`*) – mesh used to calcualte the source vectors. Source information is also defined here.
- **`phix`** (*`double NumPy array`*) – excitation fluence calculated at each node for each source. Size (NNodes, NSources)
- **`frequency`** (*`double, optional`*) – modulation frequency, in Hz. The default is 0..
- **`solver`** (*`str, optional`*) – Choose between 'CPU' or 'GPU' solver (case insensitive). Automatically determined (GPU prioritized) if not specified
- **`GPU`** (*`int, optional`*) – GPU selection. -1 for automatic, 0, 1, … for manual selection on multi-GPU systems. The default is -1.

Raises

- **`RuntimeError`** – if both CUDA and CPU versions fail.
- **`TypeError`** – if 'solver' is not 'CPU' or 'GPU'.

Returns
qvec – calculated fluence emission source vectors. Size (NNodes, NSources)

Return type
double or complex double NumPy array

nirfasterff.math.get_boundary_data

`nirfasterff.math.`**`get_boundary_data`**(*mesh*, *phi*)

Calculates boundary data given the field data in mesh

The field data can be any of the supported type: fluence, TPSF, or moments

Parameters

- **`mesh`** (*`nirfasterff mesh type`*) – the mesh whose boundary and detectors are used for the calculation.
- **`phi`** (*`double or complex double NumPy array`*) – field data as calculated by one of the 'get_field_*' solvers. Size (NNodes, NSources)

Returns
: **data** – measured boundary data at each channel. Size (NChannels,).

Return type
: double or complex double NumPy array

nirfasterff.math.get_field_CW

`nirfasterff.math.`**`get_field_CW`**(*csrI*, *csrJ*, *csrV*, *qvec*, *opt=utils.SolverOptions()*, *solver=utils.get_solver()*)

Call the Preconditioned Conjugate Gradient solver with FSAI preconditioner. For CW data only.

The current Matlab version uses COO format input, so they are NOT directly compatible

If calculation fails on GPU (if chosen), it will generate a warning and automatically switch to CPU.

On GPU, the algorithm first tries to solve for all sources simultaneously, but this can fail due to insufficient GPU memory.

If this is the case, it will generate a warning and solve the sources one by one. The latter is not as fast, but requires much less memory.

On CPU, the algorithm only solves the sources one by one.

Parameters

- **csrI** (*int32 NumPy vector, zero-based*) – I indices of the MASS matrix, in CSR format.
- **csrJ** (*int32 NumPy vector, zero-based*) – J indices of the MASS matrix, in CSR format.
- **csrV** (*double NumPy vector*) – values of the MASS matrix, in CSR format.
- **qvec** (*double NumPy array, or Scipy CSC sparse matrix*) – The source vectors. i-th column corresponds to source i. Size (NNode, NSource)

  See *gen_sources()* for details.
- **solver** (*str, optional*) – Choose between 'CPU' or 'GPU' solver (case insensitive). Automatically determined (GPU prioritized) if not specified
- **opt** (`nirfasterff.utils.SolverOptions`, *optional*) – Solver options. Uses default parameters if not specified, and they should suffice in most cases.

  See *SolverOptions()* for details

Raises

- **TypeError** – if MASS matrix and source vectors are not both real, or if solver is not 'CPU' or 'GPU'.
- **RuntimeError** – if both GPU and CPU solvers fail.

Returns

- **phi** (*double NumPy array*) – Calculated fluence at each source. Size (NNodes, Nsources)
- **info** (*nirfasterff.utils.ConvergenceInfo*) – convergence information of the solver.

  See *ConvergenceInfo()* for details

See also

gen_mass_matrix()

nirfasterff.math.get_field_FD

`nirfasterff.math.get_field_FD`(*csrI*, *csrJ*, *csrV*, *qvec*, *opt=utils.SolverOptions()*, *solver=utils.get_solver()*)

Call the Preconditioned BiConjugate Stablized solver with FSAI preconditioner.

This is designed for FD data, but can also work for CW is an all-zero imaginary part is added to the MASS matrix and source vectors.

The current Matlab version uses COO format input, so they are NOT directly compatible

If calculation fails on GPU (if chosen), it will generate a warning and automatically switch to CPU.

On GPU, the algorithm first tries to solve for all sources simultaneously, but this can fail due to insufficient GPU memory.

If this is the case, it will generate a warning and solve the sources one by one. The latter is not as fast, but requires much less memory.

On CPU, the algorithm only solves the sources one by one.

Parameters

- **csrI** (*int32 NumPy vector, zero-based*) – I indices of the MASS matrix, in CSR format.
- **csrJ** (*int32 NumPy vector, zero-based*) – J indices of the MASS matrix, in CSR format.
- **csrV** (*complex double NumPy vector*) – values of the MASS matrix, in CSR format.
- **qvec** (*complex double NumPy array, or Scipy CSC sparse matrix*) – The source vectors. i-th column corresponds to source i. Size (NNode, NSource)

  See *gen_sources()* for details.
- **solver** (*str, optional*) – Choose between 'CPU' or 'GPU' solver (case insensitive). Automatically determined (GPU prioritized) if not specified
- **opt** (nirfasterff.utils.SolverOptions, *optional*) – Solver options. Uses default parameters if not specified, and they should suffice in most cases.

  See *SolverOptions()* for details

Raises

- **TypeError** – if MASS matrix and source vectors are not both complex, or if solver is not 'CPU' or 'GPU'.
- **RuntimeError** – if both GPU and CPU solvers fail.

Returns

- **phi** (*complex double NumPy array*) – Calculated fluence at each source. Size (NNodes, Nsources)
- **info** (*nirfasterff.utils.ConvergenceInfo*) – convergence information of the solver.

  See *ConvergenceInfo()* for details

See also

gen_mass_matrix()

nirfasterff.math.get_field_TR

`nirfasterff.math.`**`get_field_TR`**(*csrI*, *csrJ*, *csrV*, *qvec*, *dt*, *max_step*, *opt=utils.SolverOptions()*, *solver=utils.get_solver()*)

Call the Preconditioned Conjugate Gradient solver with FSAI preconditioner. Calculates TPSF data

NOT interchangeable with the current MATLAB version

If calculation fails on GPU (if chosen), it will generate a warning and automatically switch to CPU.

On both GPU and CPU, the algorithm solves the sources one by one

Parameters

- **`csrI`** (*`int32 NumPy vector, zero-based`*) – I indices of the MASS matrices, in CSR format.
- **`csrJ`** (*`int32 NumPy vector, zero-based`*) – J indices of the MASS matrices, in CSR format.
- **`csrV`** (*`complex double NumPy vector`*) – values of the MASS matrices, in CSR format.

  This is calculated using gen_mass_matrix with omega=1. The real part coincides with K+C, and the imaginary part coincides with -M.

  See references for details
- **`qvec`** (*`double NumPy array, or Scipy CSC sparse matrix`*) – The source vectors. i-th column corresponds to source i. Size (NNode, NSource)

  See *`gen_sources()`* for details.
- **`dt`** (*`double`*) – time step size, in seconds.
- **`max_step`** (*`int32`*) – total number of time steps.
- **`solver`** (*`str, optional`*) – Choose between 'CPU' or 'GPU' solver (case insensitive). Automatically determined (GPU prioritized) if not specified
- **`opt`** (`nirfasterff.utils.SolverOptions`, *`optional`*) – Solver options. Uses default parameters if not specified, and they should suffice in most cases.

  See *`SolverOptions()`* for details

Raises

- **`TypeError`** – if csrV is not complex, or if qvec is not real, or if solver is not 'CPU' or 'GPU'.
- **`RuntimeError`** – if both GPU and CPU solvers fail.

Returns

- **phi** (*double NumPy array*) – Calculated TPSF at each source. Size (NNodes, Nsources*max_step), structured as,

  [src0_step0, src1_step0,. . . ,src0_step1, src1_step1,. . .]
- **info** (*nirfasterff.utils.ConvergenceInfo*) – convergence information of the solver.

  Only the convergence info of the last time step is returned.

  See *`ConvergenceInfo()`* for details

See also

`gen_mass_matrix()`

References

Arridge et al., Med. Phys,, 1993. doi:10.1118/1.597069

nirfasterff.math.get_field_TRFL

`nirfasterff.math.`**`get_field_TRFL`**(*csrI*, *csrJ*, *csrV*, *qvec_m*, *dt*, *max_step*, *opt=utils.SolverOptions()*, *solver=utils.get_solver()*)

Call the Preconditioned Conjugate Gradient solver with FSAI preconditioner. Calculates the TPSFs of fluorescence emission given the TPSFs of excitation

If calculation fails on GPU (if chosen), it will generate a warning and automatically switch to CPU.

On both GPU and CPU, the algorithm solves the sources one by one

Parameters

- **`csrI`** (*`int32 NumPy vector, zero-based`*) – I indices of the MASS matrices at emission wavelength, in CSR format.
- **`csrJ`** (*`int32 NumPy vector, zero-based`*) – J indices of the MASS matrices at emission wavelength, in CSR format.
- **`csrV`** (*`complex double NumPy vector`*) – values of the MASS matrices at emission wavelength, in CSR format.

  This is calculated using gen_mass_matrix with omega=1.
- **`qvec_m`** (*`double NumPy array`*) – TPSF of the excitation convolved with decay, and multiplied by the FEM matrix. Size (NNodes, NSources*NTime), structured as,

  [src0_step0, src1_step0,. . . ,src0_step1, src1_step1,. . .]
- **`dt`** (*`double`*) – time step size, in seconds.
- **`max_step`** (*`int32`*) – total number of time steps. It should match exactly with the number of steps in the excitation data
- **`solver`** (*`str, optional`*) – Choose between 'CPU' or 'GPU' solver (case insensitive). Automatically determined (GPU prioritized) if not specified
- **`opt`** (`nirfasterff.utils.SolverOptions`, *`optional`*) – Solver options. Uses default parameters if not specified, and they should suffice in most cases.

  See *`SolverOptions()`* for details

Raises

- **`TypeError`** – if csrV is not complex, or if phix is not real, or if solver is not 'CPU' or 'GPU'.
- **`RuntimeError`** – if both GPU and CPU solvers fail.

Returns

- **phi** (*double NumPy array*) – Calculated TPSF at each source of fluorescence emission. Size (NNodes, Nsources*max_step), structured as,

  [src0_step0, src1_step0,. . . ,src0_step1, src1_step1,. . .]
- **info** (*nirfasterff.utils.ConvergenceInfo*) – convergence information of the solver.

  Only the convergence info of the last time step is returned.

  See *`ConvergenceInfo()`* for details

See also

gen_mass_matrix(), *get_field_TR()*

nirfasterff.math.get_field_TRFLmoments

nirfasterff.math.**get_field_TRFLmoments**(*csrI*, *csrJ*, *csrV*, *csrV2*, *mx*, *gamma*, *tau*, *max_moment*, *opt=utils.SolverOptions()*, *solver=utils.get_solver()*)

Call the Preconditioned Conjugate Gradient solver with FSAI preconditioner. Directly calculates moments of re-emission using Mellin transform, given the moments of excitation

If calculation fails on GPU (if chosen), it will generate a warning and automatically switch to CPU.

On both GPU and CPU, the algorithm solves the sources one by one

Parameters

- **csrI** (*int32 NumPy vector, zero-based*) – I indices of the MASS matrices at emission wavelength, in CSR format.
- **csrJ** (*int32 NumPy vector, zero-based*) – J indices of the MASS matrices at emission wavelength, in CSR format.
- **csrV** (*complex double NumPy vector*) – values of the MASS matrices at emission wavelength, in CSR format.

  This is calculated using gen_mass_matrix with omega=1.
- **csrV2** (*double NumPy vector*) – values of the FEM integration matrix, in CSR format.

  This is calculated using gen_mass_matrix with omega=0, mua=1, kappa=0, and no boundary nodes.
- **mx** (*double NumPy array*) – moments of the excitation. Size (NNodes, Nsources*(max_moment+1)), structured as,

  [src0_m0, src1_m0,. . . ,src0_m1, src1_m1,. . .]
- **gamma** (*double NumPy array*) – defined as mesh.eta*mesh.muaf.
- **tau** (*double NumPy array*) – decay factor, as defined in mesh.tau.
- **max_moment** (*int32*) – max order of moments to calculate. That is, 0th, 1st, 2nd, .., max_moments-th will be calculated.

  This should match exact with the max_moment of the excitation
- **solver** (*str, optional*) – Choose between 'CPU' or 'GPU' solver (case insensitive). Automatically determined (GPU prioritized) if not specified
- **opt** (nirfasterff.utils.SolverOptions*, optional*) – Solver options. Uses default parameters if not specified, and they should suffice in most cases.

  See *SolverOptions()* for details

Raises

- **TypeError** – if csrV is not complex, or if mx is not real, or if solver is not 'CPU' or 'GPU'.
- **RuntimeError** – if both GPU and CPU solvers fail.

Returns

- **phi** (*double NumPy array*) – Calculated Mellin transform of fluorecence emission at each source. Size (NNodes, Nsources*(max_moment+1)), structured as,

  [src0_m0, src1_m0,. . . ,src0_m1, src1_m1,. . .]

- **info** (*nirfasterff.utils.ConvergenceInfo*) – convergence information of the solver.

  Only the convergence info of the highest order moments is returned.

  See `ConvergenceInfo()` for details

See also

`gen_mass_matrix()`, `get_field_TRmoments()`

nirfasterff.math.get_field_TRmoments

`nirfasterff.math.`**`get_field_TRmoments`**(*csrI*, *csrJ*, *csrV*, *qvec*, *max_moment*, *opt=utils.SolverOptions()*, *solver=utils.get_solver()*)

Call the Preconditioned Conjugate Gradient solver with FSAI preconditioner. Directly calculates moments of TR data using Mellin transform

NOT interchangeable with the current MATLAB version

If calculation fails on GPU (if chosen), it will generate a warning and automatically switch to CPU.

On both GPU and CPU, the algorithm solves the sources one by one

Parameters

- **csrI** (*int32 NumPy vector, zero-based*) – I indices of the MASS matrices, in CSR format.
- **csrJ** (*int32 NumPy vector, zero-based*) – J indices of the MASS matrices, in CSR format.
- **csrV** (*complex double NumPy vector*) – values of the MASS matrices, in CSR format.

  This is calculated using gen_mass_matrix with omega=1. The real part coincides with K+C, and the imaginary part coincides with -B.

  See references for details

- **qvec** (*double NumPy array, or Scipy CSC sparse matrix*) – The source vectors. i-th column corresponds to source i. Size (NNode, NSource)
- **max_moment** (*int32*) – max order of moments to calculate. That is, 0th, 1st, 2nd, .., max_moments-th will be calculated.
- **solver** (*str, optional*) – Choose between 'CPU' or 'GPU' solver (case insensitive). Automatically determined (GPU prioritized) if not specified
- **opt** (`nirfasterff.utils.SolverOptions`*, optional*) – Solver options. Uses default parameters if not specified, and they should suffice in most cases.

  See `SolverOptions()` for details

Raises

- **TypeError** – if csrV is not complex, or if qvec is not real, or if solver is not 'CPU' or 'GPU'.
- **RuntimeError** – if both GPU and CPU solvers fail.

Returns

- **phi** (*double NumPy array*) – Calculated Mellin transform at each source. Size (NNodes, Nsources*(max_moment+1)), structured as,

  [src0_m0, src1_m0,. . . ,src0_m1, src1_m1,. . .]
- **info** (*nirfasterff.utils.ConvergenceInfo*) – convergence information of the solver.

  Only the convergence info of the highest order moments is returned.

  See `ConvergenceInfo()` for details

See also

`gen_mass_matrix()`

References

Arridge and Schweiger, Applied Optics, 1995. doi:10.1364/AO.34.002683

4.1.7 nirfasterff.meshing

Functions used for mesh generation and quality check

Modules

auxiliary	Auxiliary functions used for mesh quality check.
meshutils	Functions used for mesh generation and mesh quality check

nirfasterff.meshing.auxiliary

Auxiliary functions used for mesh quality check.

They are unlikely to become useful to an ordinary user, but still documented for completeness

Use with caution: no error checking mechanisms implemented

Functions

check_facearea(nodes, ele)	Calculates the areas of each face, and check if they are close to zero
check_tetrahedron_faces(ele)	Check for faces shared by more than two tetrahedrons
checkedges(ele)	Check for orphan edges and edges shared by more than two triangles
quality_triangle_radius(nodes, ele)	Radius ratio: 2*inradius / circumradius
simpqual(nodes, ele)	For each tetrahedron, calculates the didehedral angles and returns the smallest sine of them
vector_vector_angle(u, v)	Calculates vector-vector angles, in radian

nirfasterff.meshing.auxiliary.check_facearea

`nirfasterff.meshing.auxiliary.`**`check_facearea`**(*nodes*, *ele*)

Calculates the areas of each face, and check if they are close to zero

Close to zero defined as 1e6 of the max span of the mesh

Parameters

- **nodes** (*double NumPy array*) – node locations of the mesh.
- **ele** (*int32 NumPy array*) – element list of the mesh, zero-based.

Returns

- **area** (*double NumPy vector*) – areas of each face. Size (NElements,)
- **zeroflag** (*bool NumPy vector*) – flags of whether the area is close to zero, for each face. Size (NElements,)

nirfasterff.meshing.auxiliary.check_tetrahedron_faces

`nirfasterff.meshing.auxiliary.`**`check_tetrahedron_faces`**`(`*ele*`)`

Check for faces shared by more than two tetrahedrons

Parameters
ele (*int32 NumPy array*) – element list of the mesh, zero-based.

Returns
flag – 0 if no faulty faces found, and 2 if faces shared by more than two tetrahedrons are found.

Return type
int

nirfasterff.meshing.auxiliary.checkedges

`nirfasterff.meshing.auxiliary.`**`checkedges`**`(`*ele*`)`

Check for orphan edges and edges shared by more than two triangles

Parameters
ele (*int32 NumPy array*) – element list of the mesh, zero-based.

Returns
flag – 0 if no errors found; 1 if edges shared by more than two triangles found; 2 if dangling edges found; 3 if both errors found.

Return type
int

nirfasterff.meshing.auxiliary.quality_triangle_radius

`nirfasterff.meshing.auxiliary.`**`quality_triangle_radius`**`(`*nodes*, *ele*`)`

Radius ratio: 2*inradius / circumradius

Value between 0 and 1. Equals 1 only when a triangle is equilateral

Parameters

- **nodes** (*double NumPy array*) – node locations of the mesh.
- **ele** (*int32 NumPy array*) – element list of the mesh, zero-based.

Returns
radius ratios for each triangle. Size (NElements,)

Return type
double NumPy vector

References

https://en.wikibooks.org/wiki/Trigonometry/Circles_and_Triangles/The_Incircle

nirfasterff.meshing.auxiliary.simpqual

`nirfasterff.meshing.auxiliary.`**`simpqual`**`(`*nodes*, *ele*`)`

For each tetrahedron, calculates the didehedral angles and returns the smallest sine of them

Parameters

- **nodes** (*double NumPy array*) – node locations of the mesh.
- **ele** (*int32 NumPy array*) – element list of the mesh, zero-based.

Returns
smallest sine of the dihedral angles for each element. Size (NElements,)

Return type
double NumPy vector

References

https://en.wikipedia.org/wiki/Dihedral_angle

nirfasterff.meshing.auxiliary.vector_vector_angle

`nirfasterff.meshing.auxiliary.`**`vector_vector_angle`**`(`*u*, *v*`)`

Calculates vector-vector angles, in radian

Each row of u, v is a vector, and the angles are calculated pairwise row by row

Parameters

- **u** (*double NumPy array*) – first set of vectors.
- **v** (*double NumPy array*) – second set of vectors.

Returns
pairwise vector-vector angles, in radian. Same number of rows as u and v

Return type
double NumPy vector

nirfasterff.meshing.meshutils

Functions used for mesh generation and mesh quality check

Functions

CheckMesh2D(elements, nodes[, base, verbose])	Main function that calculates and checks the quality of a 2D mesh
CheckMesh3D(elements, nodes[, base, verbose])	Main function that calculates and checks the quality of a 3D mesh, which can be either a solid or surface mesh
RunCGALMeshGenerator(mask[, opt])	Generate a tetrahedral mesh from a volume using CGAL 6.0.1 mesher, where different regions are labeled used a distinct integer.
boundfaces(nodes, elements[, base, renumber])	Finds the boundary faces of a 3D tetrahedral mesh
checkmesh3d_solid(ele, nodes[, verbose])	Calculates and returns the quality metrics of the tetrahedrons in a 3D tetrahedral mesh
checkmesh3d_surface(ele, nodes[, verbose])	Calculates and returns the quality metrics of the triangles in a 3D surface mesh

nirfasterff.meshing.meshutils.CheckMesh2D

nirfasterff.meshing.meshutils.**CheckMesh2D**(*elements*, *nodes*, *base=1*, *verbose=False*)

Main function that calculates and checks the quality of a 2D mesh

Parameters

- **elements** (*int32 NumPy array*) – element list of the mesh, zero-based.
- **nodes** (*double NumPy array*) – node locations of the mesh, in mm.
- **base** (*int, optional*) – one- or zero-based indexing of the element list. Can be 1, or 0. The default is 1.
- **verbose** (*bool, optional*) – whether print the problematic elements to stdout, if any. The default is False.

Raises

TypeError – if elements and nodes do not define a valid 2D mesh.

Returns

- **flag** (*int*) – flags of mesh quality, set by bits: 'b2b1b0'.

  b1 set if faulty edges found; b2 set if triangles with small area found
- **q_radius_ratio** (*double NumPy array*) – radius ratio of each triangle, defined as *2*inradius / circumradius*.
- **area** (*double NumPy array*) – area of each triangle in mesh.

nirfasterff.meshing.meshutils.CheckMesh3D

nirfasterff.meshing.meshutils.**CheckMesh3D**(*elements*, *nodes*, *base=1*, *verbose=False*)

Main function that calculates and checks the quality of a 3D mesh, which can be either a solid or surface mesh

If surface mesh, checkmesh3d_surface() is called

If solid mesh, checkmesh3d_solid() is first used, and checkmesh3d_surface() is subsequently used to check its outer surface

Parameters

- **ele** (*int32 NumPy array*) – element list of the mesh, zero-based.

- **nodes** (*double NumPy array*) – node locations of the mesh, in mm.
- **base** (*int, optional*) – one- or zero-based indexing of the element list. Can be 1, or 0. The default is 1.
- **verbose** (*bool, optional*) – whether print the problematic elements to stdout, if any. The default is False.

Raises

`TypeError` – if elements and nodes do not define a valid 3D mesh.

Returns

- **vol** (*double NumPy vector*) – Volume (for solid mesh) or area (for surface mesh) of each element.
- **vol_ratio** (*double NumPy vector*) – volume ratio for each tetrahedron in a solid mesh. Returns scalar 0.0 in case of a surface mesh.
- **q_area** (*double NumPy vector*) – area ratio for each triangle in a surface mesh. Returns scalar 0.0 in case of a solid mesh.
- **status_solid** (*int*) – flags of solid mesh quality, set by bits: 'b3b2b1b0'.

  b1 set if small volumes found; b2 set if small volume ratios found; b3 set if faces shared by more than two tetrahedrons found
- **status_surface** (*int*) – flags of surface mesh quality, set by bits: 'b3b2b1b0'.

  b1 set if edges shared by more than two triangles found; b2 set if dangling edges found; b3 triangles with small area found

See also

`nirfasterff.meshing.checkmesh3d_solid()`, `nirfasterff.meshing.checkmesh3d_surface()`

nirfasterff.meshing.meshutils.RunCGALMeshGenerator

`nirfasterff.meshing.meshutils.`**`RunCGALMeshGenerator`**(*mask*, *opt=utils.MeshingParams()*)

Generate a tetrahedral mesh from a volume using CGAL 6.0.1 mesher, where different regions are labeled used a distinct integer.

Internallly, the function makes a system call to the mesher binary, which can also be used standalone through the command line.

Also runs a pruning steps after the mesh generation, where nodes not referred to in the element list are removed.

Parameters

- **mask** (*uint8 NumPy array*) – 3D volumetric data defining the space to mesh. Regions defined by different integers. 0 is background.
- **opt** (`nirfasterff.utils.MeshingParams`, *optional*) – meshing parameters used. Default values will be used if not specified.

  See `nirfasterff.utils.MeshingParams()` for details

Returns

- **ele** (*int NumPy array*) – element list calculated by the mesher, one-based. Last column indicates the region each element belongs to
- **nodes** (*double NumPy array*) – element list calculated by the mesher, in mm.

References

https://doc.cgal.org/latest/Mesh_3/index.html#Chapter_3D_Mesh_Generation

nirfasterff.meshing.meshutils.boundfaces

`nirfasterff.meshing.meshutils.`**`boundfaces`**`(`*nodes*, *elements*, *base=0*, *renumber=True*`)`

Finds the boundary faces of a 3D tetrahedral mesh

Parameters

- **`nodes`** (*`double NumPy array`*) – node locations of the mesh. Size (NNodes, 3)
- **`elements`** (*`NumPy array`*) – element list of the mesh, can be either one-based or zero-based, which must be specified using the 'base' argument.
- **`base`** (*`int, optional`*) – one- or zero-based indexing of the element list. Can be 1, or 0. The default is 0.
- **`renumber`** (*`bool, optional`*) – whether renumber of the node indices in the extracted surface mesh. The default is True.

Returns

- **new_faces** (*int32 NumPy array*) – list of boundary faces of the mesh. Base of indexing is consistent with input element list

  If *renumber=True*, node indices are renumbered; if not, same node indices as in 'elements' are used

- **new_points** (*double NumPy array*) – point locations of the boundary nodes.

  If *renumber=True*, returns the subset of node loations that are on the surface; if not, it is the same as input *nodes*

nirfasterff.meshing.meshutils.checkmesh3d_solid

`nirfasterff.meshing.meshutils.`**`checkmesh3d_solid`**`(`*ele*, *nodes*, *verbose=False*`)`

Calculates and returns the quality metrics of the tetrahedrons in a 3D tetrahedral mesh

Please consider using `nirfasterff.meshing.CheckMesh3D()` instead.

Parameters

- **`ele`** (*`int32 NumPy array`*) – element list of the mesh, zero-based.
- **`nodes`** (*`double NumPy array`*) – node locations of the mesh, in mm.
- **`verbose`** (*`bool, optional`*) – whether print the problematic tetrahedrons to stdout, if any. The default is False.

Raises

`TypeError` – if mesh is not 3D tetrahedral, or if element list uses undefined nodes.

Returns

- **vol** (*double NumPy vector*) – volume of each tetrahedron, mm^3.
- **vol_ratio** (*double NumPy vector*) – volume ratio, defined as the smallest sine of dihedral angles of each tetrahedron.
- **zeroflag** (*bool NumPy vector*) – flags of whether the volume of a tetrahedron is too small.
- **faceflag** (*bool*) – whether there are faces shared by more than two tetrahedrons.

nirfasterff.meshing.meshutils.checkmesh3d_surface

nirfasterff.meshing.meshutils.**checkmesh3d_surface**(*ele*, *nodes*, *verbose=False*)

Calculates and returns the quality metrics of the triangles in a 3D surface mesh

Please consider using `nirfasterff.meshing.CheckMesh3D()` instead.

Parameters

- **ele** (*int32 NumPy array*) – element list of the mesh, zero-based.
- **nodes** (*double NumPy array*) – node locations of the mesh, in mm.
- **verbose** (*bool, optional*) – whether print the problematic triangles to stdout, if any. The default is False.

Raises

TypeError – if mesh is not 3D tetrahedral, or if element list uses undefined nodes.

Returns

- **q_radius_ratio** (*double NumPy vector*) – radius ratio of each triangle, defined as *2*inradius / circumradius*.
- **q_area_ratio** (*double NumPy vector*) – face area divided by 'ideal area' for each triangle,

  where ideal area is the area of an equilateral triangle whose edge length equals the longest edge in the face.
- **area** (*double NumPy vector*) – area of each triangle, in mm^2.
- **zeroflag** (*bool NumPy vector*) – flags whether the area a triangle is close to zero.
- **edgeflag** (*int*) – flag if any problematic edges. Flag set by bits 'b1b0':

  b1=1 if dangling edges found, b0=1 if there exist edges shared by more than two triangles

4.1.8 nirfasterff.utils

Utility functions and auxiliary classes frequently used in the package.

Functions

boundary_attenuation(n_incidence[, ...])	Calculate the boundary attenuation factor between two media.
check_element_orientation_2d(ele, nodes)	Make sure the 2D triangular elements are oriented counter clock wise.
compress_coo(coo_idx, N)	Convert COO indices to compressed.
gen_intmat_impl(mesh, xgrid, ygrid, zgrid)	YOU SHOULD NOT USE THIS FUNCTION DIRECTLY.
get_nthread()	Choose the number of OpenMP threads in CPU solvers
get_solver()	Get the default solver.
isCUDA()	Checks if system has a CUDA device with compute capability >=5.2
pointLineDistance(A, B, p)	Calculate the distance between a point and a line (defined by two points), and find the projection point
pointLocation(mesh, pointlist)	Similar to Matlab's pointLocation function, queries which elements in mesh the points belong to, and also calculate the barycentric coordinates.
pointTriangleDistance(TRI, P)	Calculate the distance between a point and a triangle (defined by three points), and find the projection point
uncompress_coo(compressed_idx)	Convert compressed indices to COO.

nirfasterff.utils.boundary_attenuation

`nirfasterff.utils.`**`boundary_attenuation`**(*n_incidence*, *n_transmission=1.0*, *method='robin'*)

Calculate the boundary attenuation factor between two media.

If vectors are used as inputs, they must have the same size and calculation is done for each pair

If n_incidence is a vector but n_transmission is a scalar, code assumes n_transmission to be the same for each value in n_incidence

Parameters

- **n_incidence** (*double Numpy vector or scalar*) – refractive index of the medium within the boundary, e.g. a tissue.
- **n_transmission** (*double Numpy vector or scalar, optional*) – refractive index of the medium outside of the boundary, e.g. air. The default is 1.0.
- **method** (*str, optional*) – boundary type, which can be,

  'robin' - internal reflectance derived from Fresnel's law

  'approx' - Groenhuis internal reflectance approximation $(1.440n^{-2} + 0.710n^{-1} + 0.668 + 0.00636n)$

  'exact' - exact internal reflectance (integrals of polarised reflectances, etc.)

  The default is 'robin'.

Raises

ValueError – if n_incidence and n_transmission are both vectors and have difference sizes, or if method is not of a recognized kind

Returns

A – calculated boundary attenuation factor.

Return type

double Numpy vector or scalar

References

Durduran et al, 2010, Rep. Prog. Phys. doi:10.1088/0034-4885/73/7/076701

nirfasterff.utils.check_element_orientation_2d

`nirfasterff.utils.`**`check_element_orientation_2d`**`(`*ele*, *nodes*`)`

Make sure the 2D triangular elements are oriented counter clock wise.

This is a direct translation from the Matlab version.

Parameters

- **ele** (*NumPy array*) – Elements in a 2D mesh. One-based. Size: (NNodes, 3).
- **nodes** (*NumPy array*) – Node locations in a 2D mesh. Size: (NNodes, 2).

Raises

TypeError – If ele does not have three rows, i.e. not a 2D triangular mesh.

Returns

ele – Re-oriented element list.

Return type

NumPy array

nirfasterff.utils.compress_coo

`nirfasterff.utils.`**`compress_coo`**`(`*coo_idx*, *N*`)`

Convert COO indices to compressed.

Parameters

- **coo_idx** (*int NumPy array*) – Input indices in COO format, zero-based.
- **N** (*int*) – Number of rows in the sparse matrix.

Returns

Output indices in compressed format, zero-based. Size (N+1,)

Return type

int NumPy array

nirfasterff.utils.gen_intmat_impl

`nirfasterff.utils.`**`gen_intmat_impl`**`(`*mesh*, *xgrid*, *ygrid*, *zgrid*`)`

YOU SHOULD NOT USE THIS FUNCTION DIRECTLY. USE MESH.GEN_INTMAT INSTEAD.

Heart of the gen_intmat function, which calculates the necessary information for converting between mesh and grid space

Parameters

- **mesh** (*nirfasterff.base.stndmesh or nirfasterff.base.fluormesh*) – The original mesh with with FEM data is calculated.
- **xgrid** (*float64 NumPy 1-D array*) – x grid in mm. Must be regular.
- **ygrid** (*float64 NumPy 1-D array*) – y grid in mm. Must be regular.
- **zgrid** (*float64 NumPy 1-D array, or [] if 2D mesh*) – z grid in mm. Must be regular.

Returns

- **gridinmesh** (*int32 NumPy array*) – Col 0: indices of grid points that are in the mesh; Col 1: indeces of the elements the grid point is in. Flattened in 'F' order, one-based.
- **meshingrid** (*int32 NumPy array*) – Indices of mesh nodes that are in the grid. One-based.
- **int_mat_mesh2grid** (*CSC sparse matrix, float64*) – Sparse matrix converting data from mesh space to grid space. Size (NGrid, NNodes).
- **int_mat_mesh2grid** (*CSC sparse matrix, float64*) – Sparse matrix converting data from grid space to mesh space. Size (NNodes, NGrid).

nirfasterff.utils.get_nthread

`nirfasterff.utils.get_nthread()`

Choose the number of OpenMP threads in CPU solvers

On CPUs with no hyperthreading, all physical cores are used Otherwise use min(physical_core, 8), i.e. no more than 8

This is heuristically determined to avoid performance loss due to memory bottlenecking

Advanced user can directly modify this function to choose the appropriate number of threads

Returns
nthread – number of OpenMP threads to use in CPU solvers.

Return type
int

nirfasterff.utils.get_solver

`nirfasterff.utils.get_solver()`

Get the default solver.

Returns
If isCUDA is true, returns 'GPU', otherwise 'CPU'.

Return type
str

nirfasterff.utils.isCUDA

`nirfasterff.utils.isCUDA()`

Checks if system has a CUDA device with compute capability >=5.2

On a Mac machine, it automatically returns False without checking

Returns
True if a CUDA device with compute capability >=5.2 exists, False if not.

Return type
bool

nirfasterff.utils.pointLineDistance

`nirfasterff.utils.pointLineDistance(A, B, p)`

Calculate the distance between a point and a line (defined by two points), and find the projection point

This is a direct translation from the Matlab version

Parameters

- `A` (*NumPy array*) – first point on the line. Size (2,) or (3,)
- `B` (*NumPy array*) – second point on the line. Size (2,) or (3,)
- `p` (*NumPy array*) – point of query. Size (2,) or (3,)

Returns

- **dist** (*double*) – point-line distance.
- **point** (*NumPy array*) – projection point on the line.

nirfasterff.utils.pointLocation

`nirfasterff.utils.pointLocation(mesh, pointlist)`

Similar to Matlab's pointLocation function, queries which elements in mesh the points belong to, and also calculate the barycentric coordinates.

This is a wrapper of the C++ function pointLocation, which implememnts an AABB tree based on Darren Engwirda's findtria package

Parameters

- `mesh` (*NIRFASTer mesh*) – Can be any of the NIRFASTer mesh types (stnd, fluor, dcs). 2D or 3D.
- `pointlist` (*NumPy array*) – A list of points to query. Shape (N, dim), where N is number of points.

Returns

- **ind** (*double NumPy array*) – i-th queried point is in element *ind[i]* of mesh (zero-based). If not in mesh, *ind[i]=-1*. Size: (N,).
- **int_func** (*double NumPy array*) – i-th row is the barycentric coordinates of i-th queried point. If not in mesh, corresponding row is all zero. Size: (N, dim+1).

References

https://github.com/dengwirda/find-tria

nirfasterff.utils.pointTriangleDistance

`nirfasterff.utils.pointTriangleDistance(TRI, P)`

Calculate the distance between a point and a triangle (defined by three points), and find the projection point

Parameters

- `TRI` (*Numpy array*) – The three points (per row) defining the triangle. Size: (3,3)
- `P` (*Numpy array*) – point of query. Size (3,).

Returns

- **dist** (*double*) – point-triangle distance.
- **PP0** (*NumPy array*) – projection point on the triangular face.

Notes

This is modified from Joshua Shaffer's code, available at: https://gist.github.com/joshuashaffer/99d58e4ccbd37ca5d96e which is based on Gwendolyn Fischer's Matlab code: https://uk.mathworks.com/matlabcentral/fileexchange/22857-distance-between-a-point-and-a-triangle-in-3d

nirfasterff.utils.uncompress_coo

`nirfasterff.utils.`**`uncompress_coo`**`(`*compressed_idx*`)`

Convert compressed indices to COO.

Parameters

`compressed_idx` (*`int NumPy array`*) – Input indices in compressed format, zero-based.

Returns

coo_idx – Output indices in COO format, zero-based.

Return type

int NumPy array

Classes

`ConvergenceInfo`([info])	Convergence information of the FEM solvers.
`MeshingParams`([xPixelSpacing, ...])	Parameters to be used by the CGAL mesher.
`SolverOptions`([max_iter, AbsoluteTolerance, ...])	Parameters used by the FEM solvers, Equivalent to 'solver_options' in the Matlab version

nirfasterff.utils.ConvergenceInfo

`class nirfasterff.utils.`**`ConvergenceInfo`**`(`*info=None*`)`

Bases: `object`

Convergence information of the FEM solvers. Only used internally as a return type of functions nirfasterff.math.get_field_*

Constructed using the output of the internal C++ functions

`isConverged`

if solver converged to relative tolerance, for each rhs

Type

bool array

`isConvergedToAbsoluteTolerance`

if solver converged to absolute tolerance, for each rhs

Type

bool array

`iteration`

iterations taken to converge, for each rhs

Type

int array

`residual`

final residual, for each rhs

Type

double array

`__init__``(`*info=None*`)`

Methods

`__init__`([info])

nirfasterff.utils.MeshingParams

class `nirfasterff.utils.MeshingParams`(*xPixelSpacing=1.*, *yPixelSpacing=1.*, *SliceThickness=1.*, *facet_angle=25.*, *facet_size=3.*, *facet_distance=2.*, *cell_radius_edge=3.*, *general_cell_size=3.*, *subdomain=np.array([0., 0.])*, *lloyd_smooth=True*, *offset=None*)

Bases: `object`

Parameters to be used by the CGAL mesher. Note: they should all be double

`xPixelSpacing`

voxel distance in x direction. Default: 1.0

Type
double

`yPixelSpacing`

voxel distance in y direction. Default: 1.0

Type
double

`SliceThickness`

voxel distance in z direction. Default: 1.0

Type
double

`facet_angle`

lower bound for the angle (in degrees) of surface facets. Default: 25.0

Type
double

`facet_size`

upper bound for the radii of surface Delaunay balls circumscribing the facets. Default: 3.0

Type
double

`facet_distance`

upper bound for the distance between the circumcenter of a surface facet and the center of its surface Delaunay ball. Default: 2.0

Type
double

`cell_radius_edge`

upper bound for the ratio between the circumradius of a mesh tetrahedron and its shortest edge. Default: 3.0

Type
double

general_cell_size

upper bound on the circumradii of the mesh tetrahedra, when no region-specific parameters (see below) are provided. Default: 3.0

Type
double

subdomain

Specify cell size for each region, in format:

```
[region_label1, cell_size1]
[region_label2, cell_size2]
    ...
```

If a region is not specified, value in "general_cell_size" will be used. Default: np.array([0., 0.])

Type
double Numpy array

lloyd_smooth

Switch for Lloyd smoother before local optimization. This can take up to 120s (hard limit set) but improves mesh quality. Default: True

Type
bool

offset

offset value to be added to the nodes after meshing. Size (3,). Defualt: None

Type
double Numpy array

Notes

Refer to CGAL documentation for details of the meshing algorithm as well as its parameters

https://doc.cgal.org/latest/Mesh_3/index.html#Chapter_3D_Mesh_Generation

`__init__`(*xPixelSpacing=1.*, *yPixelSpacing=1.*, *SliceThickness=1.*, *facet_angle=25.*, *facet_size=3.*, *facet_distance=2.*, *cell_radius_edge=3.*, *general_cell_size=3.*, *subdomain=np.array([0., 0.])*, *lloyd_smooth=True*, *offset=None*)

Methods

`__init__`([xPixelSpacing, yPixelSpacing, ...])	

nirfasterff.utils.SolverOptions

`class nirfasterff.utils.`**`SolverOptions`**(*max_iter=1000*, *AbsoluteTolerance=1e-12*, *RelativeTolerance=1e-12*, *divergence=1e8*, *GPU=-1*)

Bases: `object`

Parameters used by the FEM solvers, Equivalent to 'solver_options' in the Matlab version `max_iter`

maximum number of iterations allowed. Default: 1000

Type
int

`AbsoluteTolerance`

Absolute tolerance for convergence. Default: 1e-12

Type
double

`RelativeTolerance`

Relative (to the initial residual norm) tolerance for convergence. Default: 1e-12

Type
double

`divergence`

Stop the solver when residual norm greater than this value. Default: 1e8

Type
double

`GPU`

GPU selection. -1 for automatic, 0, 1, … for manual selection on multi-GPU systems. Default: -1

Type
int

`__init__`(*max_iter=1000*, *AbsoluteTolerance=1e-12*, *RelativeTolerance=1e-12*, *divergence=1e8*, *GPU=-1*)

Methods

`__init__`([max_iter, AbsoluteTolerance, ...])	

4.1.9 nirfasterff.visualize

Functions for basic data visualization

Functions

`plot3dmesh`(mesh[, data, selector, alpha])	Fast preview of data within a 3D FEM mesh in the nirfasterff format.
`plotimage`(mesh[, data])	Fast preview of data within a 2D FEM mesh in the nirfasterff format.

nirfasterff.visualize.plot3dmesh

`nirfasterff.visualize.`**`plot3dmesh`**(*mesh*, *data=None*, *selector=None*, *alpha=0.8*)

Fast preview of data within a 3D FEM mesh in the nirfasterff format.

Plots an image of the values on the mesh at the intersection specified by "selector".

For 2D mesh, use *`plotimage()`* instead

Parameters

- **mesh** (*nirfasterff mesh type*) – a 3D nirfasterff mesh to plot the data on.
- **data** (*double NumPy vector, optional*) – data to be plotted, with size (NNode,). If not specified, treated as all zero.
- **selector** (*str, optional*) – Specifies the intersection at which the data will be plotted, e.g. 'x>50', or '(x>50) | (y<100)', or 'x + y + z < 200'.

  Note that "=" is not supported. When "|" or "&" are used, make sure that all conditions are put in parantheses separately

  If not specified, function plots the outermost shell of the mesh.
- **alpha** (*float, optional*) – transparency, between 0-1. Default is 0.8

Raises

`TypeError` – if mesh not 2D.

Returns

- *matplotlib.figure.Figure* – the figure to be displayed
- *mpl_toolkits.mplot3d.axes3d.Axes3D* – Current axes of the plot. Can be subsequently used for further plotting.

Notes

This function is adapted from the 'plotmesh' function in the iso2mesh toolbox

https://iso2mesh.sourceforge.net/cgi-bin/index.cgi

nirfasterff.visualize.plotimage

`nirfasterff.visualize.plotimage(`*mesh*, *data=None*`)`

Fast preview of data within a 2D FEM mesh in the nirfasterff format.

Plots an image of the values on the mesh. For 3D mesh, use `plot3dmesh()` instead

Parameters

- **mesh** (*nirfasterff mesh type*) – a 2D nirfasterff mesh to plot the data on.
- **data** (*double NumPy vector, optional*) – data to be plotted, with size (NNode,). If not specified, treated as all zero.

Raises

`TypeError` – if mesh not 2D.

Returns

- *matplotlib.figure.Figure* – the figure to be displayed
- *mpl_toolkits.mplot3d.axes3d.Axes3D* – Current axes of the plot. Can be subsequently used for further plotting.

PYTHON MODULE INDEX

INDEX