NIRFASTerUFF

Release 1.2.0

Jiaming Cao, MILAB@UoB

Sep 29, 2025

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 is the compact, aka micro version of the NIRFASTerFF package.

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

FEM solver calculates: CW fluence, FD fluence

CHAPTER TWO

LINK TO THE FULL VERSION

https://github.com/milabuob/nirfaster-FF

CHAPTER THREE

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 FOUR

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 FIVE

API DOCUMENTATION

5.1 nirfasteruff

Modules

nirfasteruff.nirfasteruff
nirfasteruff.nirfasteruff_cpu
nirfasteruff.nirfasteruff_cuda

5.1.1 nirfasteruff.nirfasteruff

Classes

base()	Dummy class holding the core classes Dummy class used so the function hierarchy can be compatible with the full version
forward()	Dummy class holding the forward modeling functions.
io()	Dummy class holding some I/O functions Dummy class used so the function hierarchy can be compatible with the full version
math()	Dummy class holding some low-level functions.
meshing()	
utils()	Dummy class holding some helper functions and helper classes Dummy class used so the function hierarchy can be compatible with the full version

nirfasteruff.nirfasteruff.base

`class nirfasteruff.nirfasteruff.base`

Bases: `object`

Dummy class holding the core classes Dummy class used so the function hierarchy can be compatible with the full version

`__init__()`

Methods

`__init__`()

`class 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

`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` (`nirfasteruff.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` (`nirfasteruff.base.stndmesh`) – mesh whose .vol attribute is used to do the conversion.

Return type
None.

`class meshvol`

Bases: `object`

Small class holding the information needed for converting between mesh and volumetric space. Values calculated by nirfasteruff.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

`class 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

`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. Should be a 'stndmesh', 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 $1/\mu_s'$

Integration functions are also calculated after moving.

Parameters

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

Return type
None.

`class 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
nirfasteruff.base.optode

`meas`

information about the the detectors

Type
nirfasteruff.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
nirfasteruff.base.meshvol

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

  See `SolverOptions()` for details

Returns

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

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

  See `ConvergenceInfo()` for details

`from_copy`(*mesh*)

Deep copy all fields from another mesh.

Parameters

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

from_image(*img*, *param=utils.MeshingParams2D()*, *prop=None*, *src=None*, *det=None*, *link=None*)

Construct mesh from a segmented 2D image using JR Shewchuk's Triangle. Calls stndmesh.from_triangle after meshing step.

Parameters

- **vol** (*uint8 NumPy array*) – 2D segmented image to be meshed. 0 is considered as outside. Regions labeled using unique integers.
- **param** (nirfasteruff.utils.MeshingParams2D, *optional*) – parameters used in Triangle. If not specified, uses the default parameters defined in nirfasteruff.utils.MeshingParams2D().

  Please modify field mm_per_pixel if your image doesn't have [1,1,1] resolution

  See `MeshingParams2D()` 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** (nirfasteruff.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** (nirfasteruff.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.

References

https://www.cs.cmu.edu/~quake/triangle.html

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** (`nirfasteruff.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** (`nirfasteruff.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_triangle(*ele*, *nodes*, *prop=None*, *src=None*, *det=None*, *link=None*)

Construct a NIRFASTer mesh from a 2D triangular mesh generated by a mesher.

Can also set the optical properties and optodes if supplied

Parameters

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

  If four columns, the last column will be used for region labeling.
- **nodes** (*double NumPy array*) – node locations in the mesh. Unit: mm. Size (NNodes,2).
- **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** (`nirfasteruff.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** (`nirfasteruff.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** (`nirfasteruff.utils.MeshingParams`, *optional*) – parameters used in the CGAL mesher. If not specified, uses the default parameters defined in nirfasteruff.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** (`nirfasteruff.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** (`nirfasteruff.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 nirfasteruff.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

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.

nirfasteruff.nirfasteruff.forward

class nirfasteruff.nirfasteruff.**forward**

Bases: `object`

Dummy class holding the forward modeling functions. Dummy class used so the function hierarchy can be compatible with the full version

__init__()

Methods

__init__()	
femdata_stnd_CW([solver, opt])	Forward modeling for CW.
femdata_stnd_FD(freq[, solver, opt])	Forward modeling for FD.

`femdata_stnd_CW`(*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** (`nirfasteruff.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** (`nirfasteruff.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** (*nirfasteruff.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** (*nirfasteruff.utils.ConvergenceInfo*) – convergence information of the solver.

  See `ConvergenceInfo()` for details

See also:

`get_field_CW()`, `gen_mass_matrix()`, and `gen_sources()`

`femdata_stnd_FD`(*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** (`nirfasteruff.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** (`nirfasteruff.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** (*nirfasteruff.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** (*nirfasteruff.utils.ConvergenceInfo*) – convergence information of the solver.

  See `ConvergenceInfo()` for details

See also:

`get_field_FD()`, `gen_mass_matrix()`, and `gen_sources()`

nirfasteruff.nirfasteruff.io

`class nirfasteruff.nirfasteruff.io`

Bases: `object`

Dummy class holding some I/O functions Dummy class used so the function hierarchy can be compatible with the full version

`__init__()`

Methods

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

`readMEDIT()`

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.

`saveinr`(*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.

nirfasteruff.nirfasteruff.math

class nirfasteruff.nirfasteruff.**math**

Bases: object

Dummy class holding some low-level functions. Be careful using them: they interact closely with the C++ functions and wrong arguments used can cause unexpected crashes. Dummy class used so the function hierarchy can be compatible with the full version

__init__()

Methods

__init__()	
gen_mass_matrix(omega[, solver, GPU])	Calculate the MASS matrix, and return the coordinates in CSR format.
gen_sources()	Calculate the source vectors (point source only) for the sources in mesh.source field
get_boundary_data(phi)	Calculates boundary data given the field data in mesh
get_field_CW(csrJ, csrV, qvec[, opt, solver])	Call the Preconditioned Conjugate Gradient solver with FSAI preconditioner.
get_field_FD(csrJ, csrV, qvec[, opt, solver])	Call the Preconditioned BiConjugate Stablized solver with FSAI preconditioner.

gen_mass_matrix(*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** (nirfasteruff.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),)

`gen_sources()`

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

`get_boundary_data(`*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` (*`nirfasteruff 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

`get_field_CW(`*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** (`nirfasteruff.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** (*nirfasteruff.utils.ConvergenceInfo*) – convergence information of the solver.

  See `ConvergenceInfo()` for details

See also:

`gen_mass_matrix()`

`get_field_FD`(*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** (*nirfasteruff.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** (*nirfasteruff.utils.ConvergenceInfo*) – convergence information of the solver.

  See `ConvergenceInfo()` for details

See also:

`gen_mass_matrix()`

nirfasteruff.nirfasteruff.meshing

`class nirfasteruff.nirfasteruff.meshing`

Bases: `object`

`__init__()`

Methods

RunCGALMeshGenerator([opt])	Generate a tetrahedral mesh from a volume using CGAL 6.0.1 mesher, where different regions are labeled used a distinct integer.
__init__()	
img2mesh([opt])	Creates 2D mesh from a labeled image.

RunCGALMeshGenerator(*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`** (`nirfasteruff.utils.MeshingParams`, *`optional`*) – meshing parameters used. Default values will be used if not specified.

  See `nirfasteruff.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

`img2mesh`(*opt=utils.MeshingParams2D()*)

Creates 2D mesh from a labeled image. Using Jonathan Shewchuk's Triangle, with arguments -pPqQaA.

The different regions are labeled using continuously ascending integers starting from 1. 0 pixels are background.

Parameters

- **`img`** (*`integer Numpy array`*) – 2D image 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.MeshingParams2D()` for details

Raises

`ValueError` – if the specified max areas mismatch with number of levels, or if the labels are not continuous.

Returns

- **mesh_e** (*int NumPy array*) – element list calculated by the mesher, one-based. Last column indicates the region each element belongs to
- **mesh_n** (*double NumPy array*) – element list calculated by the mesher, in mm.

References

https://www.cs.cmu.edu/~quake/triangle.html

nirfasteruff.nirfasteruff.utils

class nirfasteruff.nirfasteruff.**utils**

Bases: `object`

Dummy class holding some helper functions and helper classes Dummy class used so the function hierarchy can be compatible with the full version

`__init__`()

Methods

`__init__`()	
`check_element_orientation_2d`(nodes)	Make sure the 2D triangular elements are oriented counter clock wise.
`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`(B, p)	Calculate the distance between a point and a line (defined by two points), and find the projection point
`pointLocation`(pointlist)	Similar to Matlab's pointLocation function, queries which elements in mesh the points belong to, and also calculate the barycentric coordinates.
`pointTriangleDistance`(P)	Calculate the distance between a point and a triangle (defined by three points), and find the projection point

class **ConvergenceInfo**(*info=None*)

Bases: `object`

Convergence information of the FEM solvers. Only used internally as a return type of functions nirfasteruff.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

class `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

class MeshingParams2D(*mm_per_pixel=1.0*, *max_area=2.0*, *offset=None*)

Bases: `object`

Parameters to be used by the Triangle mesher. Note: they should all be double

`mm_per_pixel`

pixel size in both directions, in millimeter. Default: 1.0

Type
double

`max_area`

maximum area per triangle, before scaling with mm_per_pixel. If scalar, same value is applied to all triangles

If Numpy array, it must follow this format:

```
[1 area1]
[2 area2]
...
```

where max area for each labeled region is specified individually. ALL regions must be given a value

Default: 2.0

Type
double scalar or Numpy array

`offset`

offset to be added (in mm) at the very end of the meshing (that is, after scaling). Size must be 2

Default: None, i.e. no offset to be added

Type
Numpy array or list

class `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

`check_element_orientation_2d`(*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

`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

`get_solver()`

Get the default solver.

Returns
If isCUDA is true, returns 'GPU', otherwise 'CPU'.

Return type
str

`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

`pointLineDistance`(*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.

`pointLocation`(*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 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

`pointTriangleDistance(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

PYTHON MODULE INDEX

INDEX