EMEpy

Release 1.0.0

Ian Hammond

Oct 03, 2022

CONTENTS

1 Table of Contents **1**

1.1 EMEPy Complete User Library 1

1.2 EMEPy Examples 14

2 Eigenmode Expansion **15**

3 Installation **17**

Index **19**

CHAPTER

ONE

TABLE OF CONTENTS

1.1 EMEPy Complete User Library

1.1.1 Mode

class emepy.mode.**Mode**(*x: Optional[ndarray] = None*, *y: Optional[ndarray] = None*, *wl: Optional[float] = None*, *neff: Optional[float] = None*, *Hx: Optional[ndarray] = None*, *Hy: Optional[ndarray] = None*, *Hz: Optional[ndarray] = None*, *Ex: Optional[ndarray] = None*, *Ey: Optional[ndarray] = None*, *Ez: Optional[ndarray] = None*, *n: Optional[ndarray] = None*)

Object that holds the field profiles and effective index for a 2D eigenmode

__init__(*x: Optional[ndarray] = None*, *y: Optional[ndarray] = None*, *wl: Optional[float] = None*, *neff: Optional[float] = None*, *Hx: Optional[ndarray] = None*, *Hy: Optional[ndarray] = None*, *Hz: Optional[ndarray] = None*, *Ex: Optional[ndarray] = None*, *Ey: Optional[ndarray] = None*, *Ez: Optional[ndarray] = None*, *n: Optional[ndarray] = None*) → None

Constructor for Mode Object

Parameters

- **x** (*(ndarray float)*) – array of grid points in x direction (propogation in z)
- **y** (*(ndarray float)*) – array of grid points in y direction (propogation in z)
- **wl** (*(float)*) – wavelength (meters)
- **neff** (*(float)*) – effective index
- **Hx** (*(ndarray float)*) – Hx field profile
- **Hy** (*(ndarray float)*) – Hy field profile
- **Hz** (*(ndarray float)*) – Hz field profile
- **Ex** (*(ndarray float)*) – Ex field profile
- **Ey** (*(ndarray float)*) – Ey field profile
- **Ez** (*(ndarray float)*) – Ez field profile
- **n** (*(ndarray float)*) – refractive index profile

TE_polarization_fraction()

Returns the fraction of power in the TE polarization

TM_polarization_fraction()

Returns the fraction of power in the TE polarization

effective_area()

Returns the effective area of the mode

effective_area_ratio()

Returns the ratio of the effective area to the cross-sectional area

get_confined_power(*num_pixels: Optional[int] = None*) → float

Takes in a mode and returns the percentage of power confined in the core

Parameters

num_pixels (*int*) – number of pixels outside of the core to expand the mask to capture power just outside the core (mask dilation)

Returns

Percentage of confined power

Return type

float

overlap(*m2: EigenMode*)

Returns the overlap of the mode with another mode

plot(*operation: str = 'Real'*, *colorbar: bool = True*, *normalize: bool = True*) → None

Plots the fields in the mode using pyplot. Should call plt.figure() before and plt.show() or plt.savefig() after

Parameters

- **operation** (*string or function*) – the operation to perform on the fields from ("Real", "Imaginary", "Abs", "Abs^2") (default:"Real") or a function such as np.abs
- **colorbar** (*bool*) – if true, will show a colorbar for each field
- **normalize** (*bool*) – if true, will normalize biggest field to 1

plot_material() → None

Plots the index of refraction profile

plot_power() → None

Plots the power profile

spurious_value()

Returns the spurious value of the mode

zero_phase() → None

Changes the phase such that the z components are all imaginary and the xy components are all real.

1.1.2 EME

class emepy.eme.**EME**(*layers: list = []*, *num_periods: int = 1*, *mesh_z: int = 200*, *parallel: bool = False*, *quiet: bool = False*, ***kwargs*)

The EME class is the heart of the package. It provides the algorithm that cascades sections modes together to provide the s-parameters for a geometric structure. The object is dependent on the Layer objects that are fed inside.

__init__(*layers: list = []*, *num_periods: int = 1*, *mesh_z: int = 200*, *parallel: bool = False*, *quiet: bool = False*, ***kwargs*) → None

EME class constructor

Parameters

- **layers** (*list [Layer]*) – An list of Layer objects, arranged in the order they belong geometrically. (default: [])
- **num_periods** (*int*) – Number of periods if defining a periodic structure (default: 1)
- **mesh_z** (*int*) – Number of mesh points in z per period for default monitors (default: 200)
- **parallel** (*bool*) – If true, will allocate parallelized processes for solving modes, propagating layers, and filling monitors with field data (default: False)
- **quiet** (*bool*) – If true, will not print current state and status of the solver (default: False)

add_layer(*layer: Layer*) → None

The add_layer method will add a Layer object to the EME object. The object will be geometrically added to the very right side of the structure. Using this method after propagate is useless as the solver has already been called.

Parameters

layer (Layer) – Layer object to be appended to the list of Layers inside the EME object.

add_layers(**layers*) → None

Calls add layers for the layers provided

add_monitor(*axes: str = 'xz'*, *sources: list = []*, *mesh_z: Optional[int] = None*, *z_range: Optional[tuple] = None*, *location: Optional[float] = None*, *components: Optional[list] = None*, *exempt: bool = True*) → *Monitor*

Creates a monitor associated with the eme object BEFORE the simulation is ran

Parameters

- **axes** (*str*) – the spacial axes to capture fields in. Options : 'xz' (default), 'xy', 'xz', 'xyz', 'x', 'y', 'z'. Currently only 'xz' is implemented. Note, propagation is always in z.
- **sources** (*list[Source]*) – the user can specify custom mode sources to use for this monitor (default: input left)
- **mesh_z** (*int*) – number of mesh points in z (for periodic structures, will be z * num_periods), warning: if different than global value for EME, a separate run will have to take place for this monitor (default: EME global defined)
- **z_range** (*tuple*) – tuple or list of the form (start, end) representing the range of the z values to extract
- **location** (*float*) – z coordinate where to save data for a 'xy' monitor
- **components** (*list[string]*) – a list of the field components to include. Unless the user is worried about memory, this is best left alone. (Default: ["Ex", "Ey", "Ez", "Hx", "Hy", "Hz", "n"])
- **exempt** (*bool*) – flag used for very specific case when using PML for MSEMpy. The user never has to change this value.

Returns

the newly created Monitor object

Return type

Monitor

am_master() → bool

Returns true for the master process if the user is running a parallel process using mpi. This is essential for I/O

batch_gather(*data*, *root=0*, *limit=1073741824*)

Gathers data to all workers in a batched manner that will not exceed the MPI integer limit

batch_scatter(*data*, *root=0*, *limit=1073741824*)

Scatters data to all workers in a batched manner that will not exceed the MPI integer limit

build_network() → None

Builds the full network from the cascaded layers. This is the third step in the solving process.

draw(*z_range: Optional[tuple] = None*, *mesh_z: int = 200*, *plot_sources: bool = True*, *plot_xy_sources=True*) → AxesImage

The draw method sketches a rough approximation for the xz geometry of the structure using pyplot where x is the width of the structure and z is the length. This will change in the future.

Parameters

- **z_range** (*tuple*) – tuple or list of the form (start, end) representing the range of the z values to extract
- **mesh_z** (*int*) – the number of mesh points in z to calculate index profiles for

Returns
the image used to plot the index profile

Return type
matplotlib.image.AxesImage

field_propagate(*left_coeffs: list*, *right_coeffs: list*) → None

Propagates the modes through the device to calculate the field profile everywhere

Parameters

- **left_coeffs** (*list*) – A list of floats that represent the mode coefficients for the left side of the full geometry
- **right_coeffs** (*list*) – A list of floats that represent the mode coefficients for the right side of the full geometry

get_sources() → dict

Returns a dictionary of each period and the Source objects that can be found inside each

propagate(*left_coeffs: Optional[list] = None*, *right_coeffs: list = []*) → Model

The propagate method should be called once all Layer objects have been added. This method will call the EME solver and produce s-parameters. The defulat

Parameters

- **left_coeffs** (*list*) – A list of floats that represent the mode coefficients for the left side of the full geometry. The default is determined on whether or not any custom mode sources or right_coeffs are defined. If they are, (default:[]) else (default"[1])
- **right_coeffs** (*list*) – A list of floats that represent the mode coefficients for the right side of the full geometry. (default:[])

Returns
The simphony model that represents the entire device

Return type
simphony.models.Model

propagate_layers() → None

Propagates each layer with the next by creating interface models and cascading all in parallel. This is the second step for the solver

reset(*full_reset: bool = True*, *parallel: bool = False*, *configure_parallel: bool = True*) → None

Clears out the layers and s params so the user can reuse the object in memory on a new geometry

Parameters

- **full_reset** (*boolean*) – If true, will reset everything inside of the object and allow for reinstancing without memory issues (default: True)
- **parallel** (*boolean*) – If configure_parallel is True, after reset this method will set the value of parallel. Similar to the constructor (default: False)
- **configure_parallel** (*boolean*) – If configure_parallel is True, after reset this method will set the value of parallel. Similar to the constructor (default: True)

s_parameters(*freqs=None*) → ndarray

Returns the s_parameters if they exist. If they don't exist yet, propagate() will be called first.

Returns
The s_params acquired during propagation

Return type
numpy array

solve_modes() → None

Solves for the modes in the system and is the first step in the solver's process all in parallel

class emepy.lumerical.**LumEME**(*layers=[]*, *num_periods=1*)

This class is a wrapper for EME, it performs the same operations but uses Lumerical MODE to solve for the modes at the interfaces

__init__(*layers=[]*, *num_periods=1*)

EME class constructor

Parameters

- **layers** (*list [Layer]*) – An list of Layer objects, arranged in the order they belong geometrically. (default: [])
- **num_periods** (*int*) – Number of periods if defining a periodic structure (default: 1)
- **mesh_z** (*int*) – Number of mesh points in z per period for default monitors (default: 200)
- **parallel** (*bool*) – If true, will allocate parallelized processes for solving modes, propagating layers, and filling monitors with field data (default: False)
- **quiet** (*bool*) – If true, will not print current state and status of the solver (default: False)

1.1.3 Models

class emepy.models.**Layer**(*mode_solver:* ModeSolver, *num_modes: int*, *wavelength: float*, *length: float*)

Layer objects form the building blocks inside of an EME or PeriodicEME. These represent geometric layers of rectangular waveguides that approximate continuous structures.

__init__(*mode_solver:* ModeSolver, *num_modes: int*, *wavelength: float*, *length: float*) → None

Layer class constructor

Parameters

- **mode_solver** (*Modesolver*) – ModeSolver object used to solve for the modes
- **num_modes** (*int*) – Number of total modes for the layer.
- **wavelength** (*number*) – Wavelength of eigenmode to solve for (m).
- **length** (*number*) – Geometric length of the Layer (m). The length affects the phase of the eigenmodes inside the layer via the complex phasor $e^(jz)$.

activate_layer(*sources: list = []*, *start: float = 0.0*, *period_length: float = 0.0*, *compute_modes=True*) → dict

Solves for the modes in the layer and creates an ActivatedLayer object

Parameters

- **sources** (*list[Source]*) – the Sources used to indicate where periodic layers are needed
- **start** (*number*) – the starting z value
- **periodic_length** (*number*) – the length of a single period

Returns
a dictionary that maps the period number to the activated layers. If there is no source in a period, it will be None instead at that index

Return type
dict

clear() → ndarray

Empties the modes in the ModeSolver to clear memory

Returns
the edited image

Return type
numpy array

get_activated_layer(*sources: list = []*, *start: float = 0.0*) → dict

Gets the activated layer if it exists or calls activate_layer first

Parameters
sources (*list[Source]*) – a list of Source objects for this layer

Returns
a dictionary that maps the period number to the activated layers. If there is no source in a period, it will be None instead at that index

Return type
dict

1.1.4 ModeSolver

class emepy.fd.**ModeSolver**(***kwargs*)

The ModeSolver object is the heart of finding eigenmodes for use in eigenmode expansion or simple examination. This parent class should be inherited and used as a wrapper for certain modules such as EMpy, Lumerical, Pickled data, Neural Networks, etc.

__init__(***kwargs*) → None

ModeSolver class constructor

clear() → None

Clears the modesolver's eigenmodes to make memory

get_mode(*mode_num: int*) → EigenMode

Must extract the mode of choice

Parameters
mode_num (*int*) – index of the mode of choice

solve() → None

Solves the eigenmode solver for the specific eigenmodes of desire

class emepy.lumerical.**MSLumerical**(*wl=1.55e-06*, *width=5e-07*, *thickness=2.2e-07*, *num_modes=1*, *cladding_width=5e-06*, *cladding_thickness=5e-06*, *core_index=None*, *cladding_index=None*, *mesh=300*, *mode=None*, *eme_modes=False*, *polygons=[]*, *PML=False*, ***kwargs*)

Outdated Lumerical Modesolver. Uses the lumapi Lumerical API. See Modesolver. Parameterizes the cross section as a rectangular waveguide.

__init__(*wl=1.55e-06*, *width=5e-07*, *thickness=2.2e-07*, *num_modes=1*, *cladding_width=5e-06*, *cladding_thickness=5e-06*, *core_index=None*, *cladding_index=None*, *mesh=300*, *mode=None*, *eme_modes=False*, *polygons=[]*, *PML=False*, ***kwargs*)

MSLumerical class constructor

Parameters

- **wl** (*number*) – wavelength of the eigenmodes
- **width** (*number*) – width of the core in the cross section
- **thickness** (*number*) – thickness of the core in the cross section
- **num_modes** (*int*) – number of modes to solve for (default:1)
- **cladding_width** (*number*) – width of the cladding in the cross section (default:5e-6)
- **cladding_thickness** (*number*) – thickness of the cladding in the cross section (default:5e-6)
- **core_index** (*number*) – refractive index of the core (default:Si)
- **cladding_index** (*number*) – refractive index of the cladding (default:SiO2)
- **mesh** (*int*) – number of mesh points in each direction (xy)
- **mode** (*lumapi.MODE*) – MODE object that contains the file information
- **eme_modes** (*boolean*) – if true, will utilize the lumerical eme wrapped fde solver which is not normalized to one. Produces slightly different results purely due to roundoff error during normalization.

- **PML** (*boolean*) – if true, will enable PML boundary conditions, note: this will increase the mesh and grid space

clear()

Clears the modesolver's eigenmodes to make memory

get_mode(*mode_num=0*)

Get the indexed mode number

Parameters
mode_num (*int*) – index of the mode of choice

Returns
the eigenmode of index mode_num

Return type
Mode

solve()

Solves for the eigenmodes

class emepy.fd.**MSEMpy**(*wl: float = 1.55*, *width: Optional[float] = None*, *thickness: Optional[float] = None*, *num_modes: int = 1*, *cladding_width: float = 2.5*, *cladding_thickness: float = 2.5*, *core_index: Optional[float] = None*, *cladding_index: Optional[float] = None*, *x: Optional[ndarray] = None*, *y: Optional[ndarray] = None*, *mesh: int = 128*, *accuracy: float = 1e-08*, *boundary: str = '0000'*, *epsfunc: Optional[Callable[[ndarray, ndarray], ndarray]] = None*, *n: Optional[ndarray] = None*, *PML: bool = False*, *subpixel: bool = True*, *center: tuple = (0, 0)*, ***kwargs*)

Electromagnetic Python Modesolver. Uses the EMpy library See Modesolver. Parameterizes the cross section as a rectangular waveguide.

__init__(*wl: float = 1.55*, *width: Optional[float] = None*, *thickness: Optional[float] = None*, *num_modes: int = 1*, *cladding_width: float = 2.5*, *cladding_thickness: float = 2.5*, *core_index: Optional[float] = None*, *cladding_index: Optional[float] = None*, *x: Optional[ndarray] = None*, *y: Optional[ndarray] = None*, *mesh: int = 128*, *accuracy: float = 1e-08*, *boundary: str = '0000'*, *epsfunc: Optional[Callable[[ndarray, ndarray], ndarray]] = None*, *n: Optional[ndarray] = None*, *PML: bool = False*, *subpixel: bool = True*, *center: tuple = (0, 0)*, ***kwargs*) → None

MSEMpy class constructor

Parameters

- **wl** (*number*) – wavelength of the eigenmodes
- **width** (*number*) – width of the core in the cross section
- **thickness** (*number*) – thickness of the core in the cross section
- **num_modes** (*int*) – number of modes to solve for (default:1)
- **cladding_width** (*number*) – width of the cladding in the cross section (default:5)
- **cladding_thickness** (*number*) – thickness of the cladding in the cross section (default:5)
- **core_index** (*number*) – refractive index of the core (default:Si)
- **cladding_index** (*number*) – refractive index of the cladding (default:SiO2)
- **mesh** (*int*) – number of mesh points in each direction (xy)
- **x** (*numpy array*) – the cross section grid in the x direction (z propagation) (default:None)
- **y** (*numpy array*) – the cross section grid in the y direction (z propagation) (default:None)

- **mesh** – the number of mesh points in each xy direction
- **accuracy** (*number*) – the minimum accuracy of the finite difference solution (default:1e-8)
- **boundary** (*string*) – the boundaries according to the EMpy library (default:"0000")
- **epsfunc** (*function*) – the function which defines the permittivity based on a grid (see EMpy library) (default:"0000")
- **n** (*numpy array*) – 2D profile of the refractive index
- **PML** (*bool*) – if True, will use PML boundaries. Only works for Tidy3D, not EMpy. Default : False, PEC
- **subpixel** (*bool*) – if true, will use subpixel smoothing, assuming asking for a waveguide cross section and not providing an index map (recommended)

clear() → *ModeSolver*

Clears the modesolver's eigenmodes to make memory

get_mode(*mode_num: int = 0*) → EigenMode

Get the indexed mode number

Parameters
mode_num (*int*) – index of the mode of choice

Returns
the eigenmode of index mode_num

Return type
Mode

plot_material() → None

Plots the index of refraction profile

solve() → *ModeSolver*

Solves for the eigenmodes

1.1.5 Geometry

EMEPy now offers geometry abstractions that allow users to more easily implement the layers needed for their system. This is currently under development and subject to changing. Check out emepy.geometry.py for more examples available for users.

class emepy.geometries.**Geometry**(*layers: list*)

Geoemtries are not required for users, however they do allow for easier creation of complex structures

__init__(*layers: list*) → None

Constructors should take in parameters from the user and build the layers

class emepy.geometries.**Waveguide**(*params: ~emepy.geometries.Params = <emepy.geometries.EMpyGeometryParameters object>, width: float = 0.5, thickness: float = 0.22, length: float = 1, num_modes: int = 1, center: tuple = (0, 0)*)

Block forms the simplest geometry in emepy, a single layer with a single waveguide defined

__init__(*params: ~emepy.geometries.Params = <emepy.geometries.EMpyGeometryParameters object>*, *width: float = 0.5*, *thickness: float = 0.22*, *length: float = 1*, *num_modes: int = 1*, *center: tuple = (0, 0)*) → None

Creates an instance of block which can be called to access the required layers for solving

Parameters

- **params** (*Params*) – Geometry Parameters object containing large scale parameters
- **width** (*number*) – width of the core in the cross section
- **thickness** (*number*) – thickness of the core in the cross section
- **length** (*number*) – length of the structure
- **num_modes** (*int*) – number of modes to solve for (default:1)

1.1.6 Monitors

class emepy.monitors.**Monitor**(*axes: str = 'xz'*, *dimensions: tuple = (1, 1)*, *components: list = ['E']*, *z_range: Optional[tuple] = None*, *grid_x: Optional[array] = None*, *grid_y: Optional[array] = None*, *grid_z: Optional[array] = None*, *location: Optional[float] = None*, *sources: list = []*, *adjoint_n: bool = True*, *total_length: float = 0.0*)

Monitor objects store fields during propagation for user visualization. Three types of monitors exist: 3D, 2D, and 1D.

__init__(*axes: str = 'xz'*, *dimensions: tuple = (1, 1)*, *components: list = ['E']*, *z_range: Optional[tuple] = None*, *grid_x: Optional[array] = None*, *grid_y: Optional[array] = None*, *grid_z: Optional[array] = None*, *location: Optional[float] = None*, *sources: list = []*, *adjoint_n: bool = True*, *total_length: float = 0.0*) → None

Monitor class constructor0

Parameters

- **axes** (*string*) – the spacial axes to capture fields in. Options : 'xz' (default), 'xy', 'yz', 'xyz', 'x', 'y', 'z'. Note, propagation is always in z. (default: "xy")
- **dimensions** (*tuple*) – the spacial dimensions of the resulting field (default: (1,1))
- **components** (*list*) – list of the field components to store from ('E','H','Ex','Ey','Ez','Hx','Hy','Hz) (default: ["E"])
- **z_range** (*tuple*) – tuple or list of the form (start, end) representing the range of the z values to extract (default: None)
- **grid_x** (*numpy array (default: None)*) – 1d x grid
- **grid_y** (*numpy array (default: None)*) – 1d y grid
- **grid_z** (*numpy array (default: None)*) – 1d z grid
- **location** (*float*) – the location in z if the monitor represents "xy" axes (default: None)
- **sources** (*list[Source]*) – sources to use for the monitor (default:[])
- **adjoint_n** (*bool*) – if true will use the "continuous" n used for adjoint

get_array(*component: str = 'Hy'*, *axes: Optional[str] = None*, *location: Optional[float] = None*, *z_range: Optional[tuple] = None*, *grid_x: Optional[array] = None*, *grid_y: Optional[array] = None*) → ndarray

Creates a matplotlib axis displaying the provides field component

Parameters

- **component** (*str*) – field component from "['Ex','Ey','Ez','Hx','Hy','Hz','E','H']"
- **axes** (*str*) – the spacial axes to capture fields in. Options : 'xz' (default), 'xy', 'yz', 'xyz', 'x', 'y', 'z'. Note, propagation is always in z.
- **location** (*float*) – if taken from 3D fields, users can specify where to take their 2D slice. If axes is 'xz', location refers to the location in y and 'yz' refers to a location in x and 'xy' refers to a location in z
- **z_range** (*tuple*) – tuple or list of the form (start, end) representing the range of the z values to extract
- **grid_x** (*numpy array*) – custom x grid to interpolate onto
- **grid_y** (*numpy array*) – custom y grid to interpolate onto

Returns
the requested field

Return type
numpy array

get_source_visual(*min*, *max*) → ndarray

Returns a mask with lines indicating where a source is

get_xy_monitor_visual(*min*, *max*) → ndarray

Returns a mask with lines indicating where a source is

get_z_list(*start: float*, *end: float*) → list

Finds all the points in z between start and end

Parameters

- **start** (*float*) – starting point in z
- **end** (*float*) – ending point in z

Returns
A list of tuples that take the format (i, l) where i is the index of the z point and l is the z point for all z points in the range

Return type
list[tuples]

normalize() → None

Normalizes the entire field to 1

reset_monitor() → None

Resets the fields in the monitor

visualize(*ax: Optional[AxesImage] = None*, *component: str = 'Hy'*, *axes: Optional[str] = None*, *location: float = 0*, *z_range: Optional[tuple] = None*, *show_geometry: bool = True*, *show_sources: bool = True*, *show_xy_monitors: bool = False*) → AxesImage

Creates a matplotlib axis displaying the provides field component

Parameters

- **ax** (*matplotlib axis*) – the axis object created when calling plt.figure() or plt.subplots(), if None (default) then the plt interface will be used

- **component** (*string*) – field component from "['Ex','Ey','Ez','Hx','Hy','Hz','E','H']"
- **axes** (*string*) – the spacial axes to capture fields in. Options : 'xz' (default), 'xy', 'yz', 'xyz', 'x', 'y', 'z'. Note, propagation is always in z.
- **location** (*float*) – if taken from 3D fields, users can specify where to take their 2D slice. If axes is 'xz', location refers to the location in y and 'yz' refers to a location in x and 'xy' refers to a location in z.
- **z_range** (*tuple*) – tuple or list of the form (start, end) representing the range of the z values to extract
- **show_geometry** (*bool*) – if true, will display the geometry faintly under the field profiles (default: True)
- **show_sources** (*bool*) – if true, will display a red line indicating where a source is (default: True)

Returns
the image used to plot the index profile

Return type
matplotlib.image.AxesImage

1.1.7 Neural Network Acceleration

1.1.8 Tools

EMEPy offers functions to the user that can be called. These are mostly important for the library backend however.

emepy.tools.**get_epsfunc**(*width: float*, *thickness: float*, *cladding_width: float*, *cladding_thickness: float*, *core_index: float*, *cladding_index: float*, *compute: bool = False*, *profile: np.ndarray = None*, *nx: int = None*, *ny: int = None*)

Callable class for getting epsilon on a grid

emepy.tools.**create_polygon**(*x: ndarray*, *y: ndarray*, *n: ndarray*, *detranslate: bool = True*) → list

Given a grid and a refractive index profile, will return the vertices of the polygon for importing into libraries such as Lumerical

Parameters

- **x** (*"np.ndarray"*) – the x grid
- **y** (*"np.ndarray"*) – the y grid
- **n** (*"np.ndarray"*) – the refractive index profile
- **detranslate** (*bool*) – if True, will detranslate the vertices

Returns
the resulting vertices

Return type
list[tuples]

emepy.tools.**interp**(*x: ndarray*, *y: ndarray*, *x0: ndarray*, *y0: ndarray*, *f: ndarray*, *centered: bool*) → ndarray

Interpolate a 2D complex array.

Parameters

- **x** (*"np.ndarray"*) – the new x grid

- **y** (*"np.ndarray"*) – the new y grid
- **x0** (*"np.ndarray"*) – the original x grid
- **y0** (*"np.ndarray"*) – the original y grid
- **f** (*"np.ndarray"*) – the field to interpolate
- **centered** (*bool*) – whether or not it needs to stil be shifted

Returns
the interpolated field

Return type
np.ndarray

emepy.tools.**interp1d**(*x: ndarray*, *x0: ndarray*, *f: ndarray*, *centered: bool*) → ndarray

Interpolate a 1D complex array.

Parameters

- **x** (*"np.ndarray"*) – the new grid
- **x0** (*"np.ndarray"*) – the original grid
- **f** (*"np.ndarray"*) – the field to interpolate
- **centered** (*bool*) – whether or not it needs to stil be shifted

Returns
the interpolated field

Return type
np.ndarray

emepy.tools.**into_chunks**(*location: str*, *name: str*, *chunk_size: int = 20000000*) → None

Takes a large serialized file and breaks it up into smaller chunk files

Parameters

- **location** (*string*) – the absolute or relative path of the large file
- **name** (*string*) – the name of the serialized smaller components (will have _chunk_# appended to it)
- **chunk_size** (*int*) – how big each save chunk should be

emepy.tools.**from_chunks**(*location: str*, *name: str*) → None

Takes a directory of serialized chunks that were made using into_chunks and combines them back into a large serialized file

Parameters

- **location** (*string*) – the path of the directory where the chunks are located
- **name** (*string*) – the name of the serialized file to create (make sure to include file extension if it matters)

emepy.tools.**_get_eps**(*xc: ndarray*, *yc: ndarray*, *epsfunc: Callable[[ndarray, ndarray], ndarray]*) → tuple

Used by compute_other_fields and adapted from the EMpy library

1.2 EMEPy Examples

1.2.1 Structures

- waveguide
- taper
- bragg grating
- directional coupler
- mmi
- adiabatic taper

1.2.2 Tutorials

- eigenmode solver
- Lumerical with EMEPy
- monitors
- serialize fields

EMEpy is an open-source eigenmode expansion solver implemented in Python.

Key Features

- Free and open-source
- Easy to use, great for educators
- Computationally enhancing, great for designers
- Complete design capabilities in Python

CHAPTER

TWO

EIGENMODE EXPANSION

Eigenmode Expansion (EME) is a method of simulating light through optical structures that operates in the frequency domain. The algorithm works by utilizing some useful properties of light. First, light exists as a superposition of eigenmodes that satisfy Maxwell's equations inside the structure. The eigenmodes are composed of a field pattern and an eigenvalue, $beta$ proportional to the effective index of refraction of the structure. As these eigenmodes propagate through a structure that changes shape or material along the direction of propagation, the effective index and field patterns change. However, if the structure does not change in this direction, the eigenmodes remain the same except for the phase. Along these structures, the phase changes according to $e^{jbeta z}$ where z is the distance travelled.

The EME algorithm utilizes this property by taking geometric structures and representing them as a series of continuous structures in the direction of propagation. This way, each section of the geometry can contain an set of eigenmodes and phase changes. At each intersection between sections, the power of the input eigenmodes transfer into the power of the output eigenmodes. However, unless the to sets of modes are identical, reflection can also occur.

To calculate the proportion of power that transmits and reflects from any given mode to another, the overlap is calculated and a system of equations is solved. Together, the intersection mode overlap and phase propagation are cascaded and provide a set of s-parameters for the device. EMEpy can be used to calculate these values and produce s-parameters for users' geometry.

CHAPTER

THREE

INSTALLATION

EMEpy can be found on pip.

```
pip install emepy
```

For the latest version, the source code can be found on GitHub. Clone the directory onto your local desktop with:

```
git clone --depth 1 git@github.com:BYUCamachoLab/emepy.git
```

Then install from within the repo:

```
pip install -e .
```

To install the neural network models:

> Read these instructions

INDEX

Symbols

__init__() (*emepy.eme.EME method*), 2
__init__() (*emepy.fd.MSEMpy method*), 8
__init__() (*emepy.fd.ModeSolver method*), 7
__init__() (*emepy.geometries.Geometry method*), 9
__init__() (*emepy.geometries.Waveguide method*), 9
__init__() (*emepy.lumerical.LumEME method*), 5
__init__() (*emepy.lumerical.MSLumerical method*), 7
__init__() (*emepy.mode.Mode method*), 1
__init__() (*emepy.models.Layer method*), 6
__init__() (*emepy.monitors.Monitor method*), 10
_get_eps() (*in module emepy.tools*), 13

A

activate_layer() (*emepy.models.Layer method*), 6
add_layer() (*emepy.eme.EME method*), 3
add_layers() (*emepy.eme.EME method*), 3
add_monitor() (*emepy.eme.EME method*), 3
am_master() (*emepy.eme.EME method*), 3

B

batch_gather() (*emepy.eme.EME method*), 4
batch_scatter() (*emepy.eme.EME method*), 4
build_network() (*emepy.eme.EME method*), 4

C

clear() (*emepy.fd.ModeSolver method*), 7
clear() (*emepy.fd.MSEMpy method*), 9
clear() (*emepy.lumerical.MSLumerical method*), 8
clear() (*emepy.models.Layer method*), 6
create_polygon() (*in module emepy.tools*), 12

D

draw() (*emepy.eme.EME method*), 4

E

effective_area() (*emepy.mode.Mode method*), 2
effective_area_ratio() (*emepy.mode.Mode method*), 2
EME (*class in emepy.eme*), 2

F

field_propagate() (*emepy.eme.EME method*), 4
from_chunks() (*in module emepy.tools*), 13

G

Geometry (*class in emepy.geometries*), 9
get_activated_layer() (*emepy.models.Layer method*), 6
get_array() (*emepy.monitors.Monitor method*), 10
get_confined_power() (*emepy.mode.Mode method*), 2
get_epsfunc() (*in module emepy.tools*), 12
get_mode() (*emepy.fd.ModeSolver method*), 7
get_mode() (*emepy.fd.MSEMpy method*), 9
get_mode() (*emepy.lumerical.MSLumerical method*), 8
get_source_visual() (*emepy.monitors.Monitor method*), 11
get_sources() (*emepy.eme.EME method*), 4
get_xy_monitor_visual() (*emepy.monitors.Monitor method*), 11
get_z_list() (*emepy.monitors.Monitor method*), 11

I

interp() (*in module emepy.tools*), 12
interp1d() (*in module emepy.tools*), 13
into_chunks() (*in module emepy.tools*), 13

L

Layer (*class in emepy.models*), 6
LumEME (*class in emepy.lumerical*), 5

M

Mode (*class in emepy.mode*), 1
ModeSolver (*class in emepy.fd*), 7
Monitor (*class in emepy.monitors*), 10
MSEMpy (*class in emepy.fd*), 8
MSLumerical (*class in emepy.lumerical*), 7

N

normalize() (*emepy.monitors.Monitor method*), 11

O

overlap() (*emepy.mode.Mode method*), 2

P

plot() (*emepy.mode.Mode method*), 2
plot_material() (*emepy.fd.MSEMpy method*), 9
plot_material() (*emepy.mode.Mode method*), 2
plot_power() (*emepy.mode.Mode method*), 2
propagate() (*emepy.eme.EME method*), 4
propagate_layers() (*emepy.eme.EME method*), 5

R

reset() (*emepy.eme.EME method*), 5
reset_monitor() (*emepy.monitors.Monitor method*), 11

S

s_parameters() (*emepy.eme.EME method*), 5
solve() (*emepy.fd.ModeSolver method*), 7
solve() (*emepy.fd.MSEMpy method*), 9
solve() (*emepy.lumerical.MSLumerical method*), 8
solve_modes() (*emepy.eme.EME method*), 5
spurious_value() (*emepy.mode.Mode method*), 2

T

TE_polarization_fraction() (*emepy.mode.Mode method*), 1
TM_polarization_fraction() (*emepy.mode.Mode method*), 1

V

visualize() (*emepy.monitors.Monitor method*), 11

W

Waveguide (*class in emepy.geometries*), 9

Z

zero_phase() (*emepy.mode.Mode method*), 2