1 2D Crossing Optimization 1
2 LumOpt’s Documentation Table of Content 3
3 Indices and tables 19
Python Module Index 21
CHAPTER 1

2D Crossing Optimization
2.1 Basic Theory

Notes to come.

2.1.1 Resources

Some resources for understanding Adjoint based Photonics optimization are:

Papers

- Adjoint shape optimization applied to electromagnetic design [Link](https://www.osapublishing.org/oe/abstract.cfm?uri=oe-21-18-21693)
- Shape optimization of nanophotonic devices using the adjoint method [Link](http://ieeexplore.ieee.org/document/6989970/)

Dissertations

- Owen Miller (Berkeley, Yablonovitch group): Photonic Design: From Fundamental Solar Cell Physics to Computational Inverse Design [Link](http://optoelectronics.eecs.berkeley.edu/ThesisOwenMiller.pdf) Focuses on the continuous adjoint method for electromagnetics, and has some very good intuitive explanations.
LumOpt Documentation

- Vidya Ganapati (Berkeley, Yablonovitch group): Optical Design Considerations for High Conversion Efficiency in Photovoltaics
- Christopher Lalau-Keraly (Berkeley, Yablonovitch group): Optimizing Nanophotonics: from Photoreceivers to Waveguides
  https://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-20.pdf Applications for Silicon Photonics

2.2 Installation and First Optimization

2.2.1 Install

LumOpt runs on python 2.7, with Lumerical xx or better.

Choose your install directory and run

```
git clone https://github.com/chriskeraly/LumOpt.git
python setup.py -develop
```

You will need to add the Lumerical API `lumapi` to your Python path.

2.2.2 Running a prebuilt optimization: a 2D Silicon Photonics Waveguide Y-branch

My favorite way of running optimizations is from a jupyter notebook, that way, you can inspect the results in detail after the optimization, keep a record of the results, or debug the optimization if need be.

In that case just copy the contents of `examples/splitter/splitter_opt.py` into a notebook and run it.

From the terminal:

```
cd examples/splitter
python splitter_opt_2D.py
```

Or run the file from your favorite IDE.

If everything is installed correctly, you should see Lumerical windows open, and eventually you should see:

2.3 Tutorial

2.3.1 Splitter Optimization

Let’s have a look at the code in `examples/splitter/splitter_opt.py`

```python
""" Copyright chriskeraly
   Copyright (c) 2019 Lumerical Inc. """

######## IMPORTS ########
# General purpose imports
import os
import numpy as np
import scipy as sp

# Optimization specific imports

(continues on next page)"""
from lumopt.utilities.load_lumerical_scripts import load_from_lsf
from lumopt.utilities.wavelengths import Wavelengths
from lumopt.geometries.polygon import FunctionDefinedPolygon
from lumopt.figures_of_merit.modematch import ModeMatch
from lumopt.optimizers.generic_optimizers import ScipyOptimizers
from lumopt.optimization import Optimization

######## DEFINE BASE SIMULATION ########
base_script = load_from_lsf(os.path.join(os.path.dirname(__file__), 'splitter_base_TE_modematch.lsf'))

######## DEFINE SPECTRAL RANGE #########
# Global wavelength/frequency range for all the simulations
wavelengths = Wavelengths(start = 1300e-9, stop = 1800e-9, points = 21)

######## DEFINE OPTIMIZABLE GEOMETRY ########
# The class FunctionDefinedPolygon needs a parameterized Polygon (with points ordered
# in a counter-clockwise direction). Here the geometry is defined by 10 parameters
# defining the knots of a spline, and the resulting Polygon has 200 edges, making it quite
# smooth.
initial_points_x = np.linspace(-1.0e-6, 1.0e-6, 10)
initial_points_y = 0.25e-6 + (0.6e-6 - 0.25e-6) * np.power(np.sin(np.pi / 2.0 * (initial_points_x - initial_points_x.min())) / (initial_points_x.max() - initial_points_x.min()), 2)
def taper_splitter(params = initial_points_y):
    ''' Defines a taper where the parameters are the y coordinates of the nodes of a
    cubic spline. '''
    points_x = np.concatenate(([initial_points_x.min() - 0.01e-6], initial_points_x, [initial_points_x.max() + 0.01e-6]))
    points_y = np.concatenate(([initial_points_y.min()], params, [initial_points_y.max()]))
    n_interpolation_points = 100
    polygon_points_x = np.linspace(min(points_x), max(points_x), n_interpolation_points)
    interpolator = sp.interpolate.interpld(points_x, points_y, kind = 'cubic')
    polygon_points_y = interpolator(polygon_points_x)
    polygon_points_up = [(x, y) for x, y in zip(polygon_points_x, polygon_points_y)]
    polygon_points_down = [(x, -y) for x, y in zip(polygon_points_x, polygon_points_y)]
    polygon_points = np.array(polygon_points_up[::-1] + polygon_points_down)
    return polygon_points

# The geometry will pass on the bounds and initial parameters to the optimizer.
bounds = [(0.2e-6, 0.8e-6)] * initial_points_y.size
# The permittivity of the material making the optimizable geometry and the
# permittivity of the material surrounding
# it must be defined. Since this is a 2D simulation, the depth has no importance. The
# edge precision defines the
# discretization of the edges forming the optimizable polygon. It should be set such
# there are at least a few points
# per mesh cell. An effective index of 2.8 is user to simulate a 2D slab of 220 nm
# thickness.
geometry = FunctionDefinedPolygon(func = taper_splitter, initial_params = initial_points_y, bounds = bounds, z = 0.0, depth = 220e-9, eps_out = 1.44 ** 2, eps_in = 2.8 ** 2, edge_precision = 5, dx = 0.1e-9)

(continues on next page)
2.3.2 Optimization Philosophy

The optimization setup works as such:

- The user must provide a Lumerical script that serves as the basis of the optimization. It has everything required to build the problem except for the optimizable geometry, which is defined later in the python script and added in the simulation by the optimization itself.

- The user then defines in python, using the classes of LumOpt:
  - An optimizable geometry
  - A Figure of Merit
  - An Optimization algorithm

2.3.3 Lumerical Setup Script

Here is the setup script examples/Ysplitter/splitter_base_TE_modematch.lsf used in this example:

```python
switchtolayout;
selectall;
delete;

## SIM PARAMS
size_x  = 3.0e-6;
size_y  = 3.0e-6;
mesh_x  = 20.0e-9;
mesh_y  = 25.0e-9;
finer_mesh_size_x = 2.5e-6;
finer_mesh_size_y = 2.5e-6;
background_index = 1.44;
mesh_accuracy = 1;
```

(continues on next page)
## GEOMETRY

wg_width = 0.5e-6;
wg_length = 3.0e-6;
wg_height = 220.0e-9;
wg_material = '<Object defined dielectric>,'
wg_ref_index = 2.8;

#INPUT WAVEGUIDE
addrect;
set('name','input wg');
set('x span',wg_length);
set('y span',wg_width);
set('z span',wg_height);
set('y',0.0);
set('x',-2.5e-6);
set('index',wg_ref_index);
set('material',wg_material);

#OUTPUT WAVEGUIDES
addrect;
set('name','output wg top');
set('x span',wg_length);
set('y span',wg_width);
set('z span',wg_height);
set('y',0.35e-6);
set('x',2.5e-6);
set('index',wg_ref_index);
set('material',wg_material);
addrect;
set('name','output wg bottom');
set('x span',wg_length);
set('y span',wg_width);
set('z span',wg_height);
set('y',-0.35e-6);
set('x',2.5e-6);
set('index',wg_ref_index);
set('material',wg_material);

## SOURCE
addmode;
set('direction','Forward');
set('injection axis','x-axis');
set('y',0.0);
set('y span',finer_mesh_size_y);
set('x',-1.25e-6);
set('override global source settings',false);
set('mode selection','fundamental TE mode');

## FDTD
addfdtd;
set('dimension','2D');
set('background index',background_index);
set('mesh accuracy',mesh_accuracy);
set('x min',-size_x/2.0);
set('x max',size_x/2.0);
set('y min',-size_y/2.0);
set('y max',size_y/2.0);
set('force symmetric y mesh',true);

## MESH IN OPTIMIZABLE REGION
addmesh;
set('directly defined',true);
set('x',0.0);
set('x span',finer_mesh_size_x);
set('y',0.0);
set('y span',finer_mesh_size_y);
set('set maximum mesh step',true);
set('override x mesh',true);
set('dx',mesh_x);
set('override y mesh',true);
set('dy',mesh_y);
set('override z mesh',false);

## OPTIMIZATION FIELDS MONITOR IN OPTIMIZABLE REGION
addpower;
set('name','opt_fields');
set('monitor type','2D Z-normal');
set('x',0.0);
set('x span',finer_mesh_size_x);
set('y',0.0);
set('y span',finer_mesh_size_y);

## FOM FIELDS
addpower;
set('name','fom');
set('monitor type','2D X-normal');
set('x',1.25e-6);
set('y',0.0);
set('y span',finer_mesh_size_y);

If the setup script is run in Lumerical FDTD (or alternatively replace `opt.run()` with `opt.make_base_sim()` in the previous code snippet, here is a screenshot of what is built:
As one can see everything but the optimizable geometry is present. There are some additional requirements to this base simulation however, beyond the strict minimum to perform the simulation.

**Base Simulation Requirements:**

**SOURCES:**

Sources MUST have ‘source’ in their name. This is because to run the adjoint simulation, all the classic forward simulation sources will have to be deleted. The easiest way to do this is to identify by their name. IF YOU HAVE A SOURCE WITHOUT ‘source’, IN ITS NAME YOUR OPTIMIZATION WILL MOST LIKELY FAIL.

**OPTIMIZATION FIELD MONITOR:**

A field Monitor named ‘opt_fields’ must cover the entire region where your optimizable geometry is susceptible to go. This is because the fields in your optimizable geometry must be known in order to calculate the shape derivatives. Make sure that even once the optimization gets going, your geometry will not go beyond this box. If it does the optimization will crash. Be particularly careful with splines, which can sometimes shoot well beyond their knots.

**MESH:**

In the same way as the optimization field monitor above, a mesh refinement region should be placed over the region where the geometry is susceptible to go. This is to ensure that the mesh does not change during the optimization, which would also cause the optimization to crash.

### 2.3.4 Co-optimization

The + operator can be used between two optimization objects that use the same parameters. This is for the moment the only way to do multi-wavelength optimization for example.

An example of this can be seen in `examples/Ysplitter/robust_coupler.py` where a the performance of a coupler is optimized for robustness: the figure of merit is the sum of the figures of merit for the nominal geometry, and a
geometry with a 25nm extra bias.

```python
# Define spectral range
wavelengths = Wavelengths(start = 1550e-9, stop = 1550e-9, points = 1)

# Define base simulation
script_1 = load_from_lsf(os.path.join(os.path.dirname(__file__), 'splitter_base_TE_modematch.lsf'))
script_2 = load_from_lsf(os.path.join(os.path.dirname(__file__), 'splitter_base_TE_modematch_25nmoffset.lsf'))

# Define optimizable geometry
def taper_splitter_1(params = initial_points_y):
    points_x = np.concatenate(([initial_points_x.min() - 0.01e-6], initial_points_x, [initial_points_x.max() + 0.01e-6]))
    points_y = np.concatenate(([initial_points_y.min()], params, [initial_points_y.max()]))
    n_interpolation_points = 100
    polygon_points_x = np.linspace(min(points_x), max(points_x), n_interpolation_points)
    interpolator = sp.interpolate.interp1d(points_x, points_y, kind = 'cubic')
    polygon_points_up = [(x, y) for x, y in zip(polygon_points_x, polygon_points_y)]
    polygon_points_down = [(x, -y) for x, y in zip(polygon_points_x, polygon_points_y)]
    polygon_points = np.array(polygon_points_up[::-1] + polygon_points_down)
    return polygon_points

dy = 25.0e-9
def taper_splitter_2(params = initial_points_y + dy):
    points_x = np.concatenate(([initial_points_x.min() - 0.01e-6], initial_points_x, [initial_points_x.max() + 0.01e-6]))
    points_y = np.concatenate(([initial_points_y.min() + dy], params, [initial_points_y.max() + dy]))
```

(continues on next page)
n_interpolation_points = 100
polygon_points_x = np.linspace(min(points_x), max(points_x), n_interpolation_points)
interpolator = sp.interpolate.interp1d(points_x, points_y, kind = 'cubic')
polygon_points_up = [(x, y) for x, y in zip(polygon_points_x, polygon_points_y)]
polygon_points_down = [(x, -y) for x, y in zip(polygon_points_x, polygon_points_y)]
polygon_points = np.array(polygon_points_up[::-1] + polygon_points_down)
return polygon_points

bounds = [(0.2e-6, 0.9e-6)] * initial_points_y.size
# guess from splitter_opt_2D.py optimization
geometry_1 = FunctionDefinedPolygon(func = taper_splitter_1, initial_params = initial_points_y, bounds = bounds, z = 0.0, depth = 220e-9, eps_out = 1.44 ** 2, eps_in = 2.8 ** 2, edge_precision = 5, dx = 0.1e-9)
geometry_2 = FunctionDefinedPolygon(func = taper_splitter_2, initial_params = initial_points_y + dy, bounds = bounds, z = 0.0, depth = 220e-9, eps_out = 1.44 ** 2, eps_in = 2.8 ** 2, edge_precision = 5, dx = 0.1e-9)

######## DEFINE FIGURE OF MERIT ########
# Although we are optimizing for the same thing, two separate fom objects must be create
def_fom_1 = ModeMatch(monitor_name = 'fom', mode_number = 3, direction = 'Forward',
target_T_fwd = lambdawl: np.ones(wl.size), norm_p = 1)
def_fom_2 = ModeMatch(monitor_name = 'fom', mode_number = 3, direction = 'Forward',
target_T_fwd = lambdawl: np.ones(wl.size), norm_p = 1)

######## DEFINE OPTIMIZATION ALGORITHM ########
#Eventually this will be improved
optimizer_1 = ScipyOptimizers(max_iter = 40, method = 'L-BFGS-B', scaling_factor = 1e6, pgtol = 1e-9)
optimizer_2 = ScipyOptimizers(max_iter = 40, method = 'L-BFGS-B', scaling_factor = 1e6, pgtol = 1e-9)

######## PUT EVERYTHING TOGETHER ########
opt_1 = Optimization(base_script = script_1, wavelengths = wavelengths, fom = fom_1,
geometry = geometry_1, optimizer = optimizer_1, hide_fdtd_cad = False, use_deps = True)
opt_2 = Optimization(base_script = script_2, wavelengths = wavelengths, fom = fom_2,
geometry = geometry_2, optimizer = optimizer_2, hide_fdtd_cad = False, use_deps = True)
opt = opt_1 + opt_2

######## RUN THE OPTIMIZER ########
opt.run()
2.4 Examples

2.4.1 2D Y-Splitter Optimization

2.4.2 2D Crossing Optimization

2.4.3 2D Grating Coupler

2.5 User Classes and Functions

2.5.1 Optimization

2.5.2 Figures of Merit

class lumopt.figures_of_merit.mode_match.ModeMatch(monitor_name, mode_number, direction, target_T_fwd=<function <lambda>>, norm_p=1)

Calculates the figure of merit from an overlap integral between the fields recorded by a field monitor and the selected mode. A mode expansion monitor is added to the field monitor to calculate the overlap result, which appears as T_forward in the list of mode expansion monitor results. The T_forward result is described in the following page:

https://kb.lumerical.com/ref_sim_obj_using_mode_expansion_monitors.html

This result is equivalent to equation (7) in the following paper:


Parameters

• monitor_name – name of the field monitor that records the fields to be used in the mode overlap calculation.
• mode_number – mode number in the list of modes generated by the mode expansion monitor.
• direction – direction of propagation (‘Forward’ or ‘Backward’) of the mode injected by the source.
• target_T_fwd – function describing the target T_forward vs wavelength (see documentation for mode expansion monitors).
• norm_p – exponent of the p-norm used to generate the figure of merit; use to generate the FOM.

2.5.3 Geometries
class lumopt.geometries.polygon.Polygon(points, z, depth, eps_out, eps_in, edge_precision)

Defines a polygon with vertices on the (x,y)-plane that are extruded along the z direction to create a 3-D shape. The vertices are defined as a numpy array of coordinate pairs np.array([[x0,y0],...,[xn,yn]]). THE VERTICES MUST BE ORDERED IN A COUNTER CLOCKWISE DIRECTION.

Parameters
• **points** – array of shape (N,2) defining N polygon vertices.
• **z** – center of polygon along the z-axis.
• **depth** – span of polygon along the z-axis.
• **eps_out** – permittivity of the material around the polygon.
• **eps_in** – permittivity of the polygon material.
• **edge_precision** – number of quadrature points along each edge for computing the FOM gradient using the shape derivative approximation method.

```python
class lumopt.geometries.polygon.FunctionDefinedPolygon(func, initial_params, bounds, z, depth, eps_out, eps_in, edge_precision=5, dx=1e-10)
```

Constructs a polygon from a user defined function that takes the optimization parameters and returns a set of vertices defining a polygon. The polygon vertices returned by the function must be defined as a numpy array of coordinate pairs np.array([(x0,y0),...,(xn,yn)]). THE VERTICES MUST BE ORDERED IN A COUNTER CLOCKWISE DIRECTION.

**Parameters**

• **fun** – function that takes the optimization parameter values and returns a polygon.
• **initial_params** – initial optimization parameter values.
• **bounds** – bounding ranges (min/max pairs) for each optimization parameter.
• **z** – center of polygon along the z-axis.
• **depth** – span of polygon along the z-axis.
• **eps_out** – permittivity of the material around the polygon.
• **eps_in** – permittivity of the polygon material.
• **edge_precision** – number of quadrature points along each edge for computing the FOM gradient using the shape derivative approximation method.
• **dx** – step size for computing the FOM gradient using permittivity perturbations.

### 2.5.4 Optimizers

```python
class lumopt.optimizers.generic_optimizers.ScipyOptimizers(max_iter, method='L-BFGS-G', scaling_factor=1.0, pgtol=1e-05)
```

Wrapper for the optimizers in SciPy’s optimize package:

https://docs.scipy.org/doc/scipy/reference/optimize.html#module-scipy.optimize

Some of the optimization algorithms available in the optimize package (‘L-BFGS-G’ in particular) can approximate the Hessian from the different optimization steps (also called Quasi-Newton Optimization). While this is very powerful, the figure of merit gradient calculated from a simulation using a continuous adjoint method can be noisy. This can point Quasi-Newton methods in the wrong direction, so use them with caution.

**Parameters**

• **max_iter** – maximum number of iterations; each iteration can make multiple figure of merit gradient evaluations.
• **method** – string with the chosen minimization algorithm.
• **scaling_factor** – optimization parameters are scaled by this factor.

• **pgtol** – gradient tolerance parameter ‘gtol’ (see ‘BFGS’ or ‘L-BFGS-G’ documentation).

```python
class lumopt.optimizers.fixed_step_gradient_descent.FixedStepGradientDescent(max_dx, max_iter, all_params_equal, noise_magnitude, scaling_factor)
```

Gradient descent with the option to add noise and a parameter scaling. The update equation is:

\[ \Delta p_i = \frac{\frac{d\text{FOM}}{dp_i}}{\max_j(|\frac{d\text{FOM}}{dp_j}|)} \Delta x + \text{ext\{noise\}_i} \]

If all_params_equal = True, then the update equation is:

\[ \Delta p_i = \text{sign}(\frac{d\text{FOM}}{dp_i}) \Delta x + \text{ext\{noise\}_i} \]

If the optimization has many local optima: \(\text{noise} = \text{rand([-1,1])} \times \text{noise\_magnitude}\).

- **param max_dx** maximum allowed change of a parameter per iteration.
- **param max_iter** maximum number of iterations to run.
- **param all_params_equal** if true, all parameters will be changed by +/- dx depending on the sign of their associated shape derivative.
- **param noise_magnitude** amplitude of the noise.
- **param scaling_factor** scaling factor to bring the optimization variables closer to 1

```python
class lumopt.optimizers.adaptive_gradient_descent.AdaptiveGradientDescent(max_dx, min_dx, max_iter, dx_regrowth_factor, all_params_equal, scaling_factor)
```

Almost identical to FixedStepGradientDescent, except that dx changes according to the following rule:

\[ dx = \min(\max_dx, dx \times dx\_regrowth\_factor) \text{ while } \text{newfom} < \text{oldfom} dx = dx / 2 \text{ if } dx < \min_dx: \]

\[ dx = \min_dx \text{ return newfom} \]

**Parameters**

- **max_dx** – maximum allowed change of a parameter per iteration.
- **min_dx** – minimum step size (for the largest parameter changing) allowed.
- **dx_regrowth_factor** – by how much dx will be increased at each iteration.
- **max_iter** – maximum number of iterations to run.
- **all_params_equal** – if true, all parameters will be changed by +/- dx depending on the sign of their associated shape derivative.
- **scaling_factor** – scaling factor to bring the optimization variables closer to one.
2.5.5 Materials

```python
class lumopt.utilities.materials.Material(base_epsilon=1.0, name='<Object defined dielectric>', mesh_order=None)
```

Permittivity of a material associated with a geometric primitive. In FDTD Solutions, a material can be given in two ways:

1. By providing a material name from the material database (e.g. ‘Si (Silicon) - Palik’) that can be assigned to a geometric primitive.
2. By providing a refractive index value directly in geometric primitive.

To use the first option, simply set the name to ‘<Object defined dielectric>’ and enter the desired base permittivity value. To use the second option, set the name to the desired material name (base permittivity will be ignored).

**Parameters**

- `name` – string (such as ‘Si (Silicon) - Palik’) with a valid material name.
- `base_epsilon` – scalar base permittivity value.
- `mesh_order` – order of material resolution for overlapping primitives.

2.5.6 Helpers

Copyright chriskeraly Copyright (c) 2019 Lumerical Inc.

```python
lumopt.utilities.load_lumerical_scripts.load_from_lsf(script_file_name)
```

Loads the provided script as a string and strips out all comments.

**Parameters**

- `script_file_name` – string specifying a file name.

2.6 Internal functions

These are not meant to be used directly when running optimizations

2.6.1 Fields

```python
class lumopt.utilities.fields.Fields(x, y, z, wl, E, D, eps, H)
```

Container for the raw fields from a field monitor. Several interpolation objects are created internally to evaluate the fields at any point in space. Use the auxiliary :method:lumopt.lumerical_methods.lumerical_scripts.get_fields to create this object.

2.6.2 Gradient Fields

```python
class lumopt.utilities.gradients.GradientFields(forward_fields, adjoint_fields)
```

Combines the forward and adjoint fields (collected by the constructor) to generate the integral used to compute the partial derivatives of the figure of merit (FOM) with respect to the shape parameters.

```python
boundary_perturbation_integrand()
```

Generates the integral kernel in equation 5.28 of Owen Miller’s thesis used to approximate the partial derivatives of the FOM with respect to the optimization parameters.
**spatial_gradient_integral** *(d_epses)*

Uses permittivity gradients to calculate the gradient of the figure of merit (FOM) with respect to each of the optimization parameters. The permittivity gradients can be obtained from FDTD.

**D_epses** list of arrays containing the partial derivatives of the permittivity w.r.t. each optimization parameter.

### 2.6.3 Others

Copyright chriskeraly Copyright (c) 2019 Lumerical Inc.

lumopt.utilities.load_lumerical_scripts.load_from_lsf(script_file_name)

Loads the provided script as a string and strips out all comments.

**Parameters**

- **script_file_name** – string specifying a file name.

**class** lumopt.utilities.simulation.Simulation(workingDir, hide_fdtdCad)

Object to manage the FDTD CAD.

**Parameters**

- **workingDir** – working directory for the CAD session.
- **hide_fdtdCad** – if true, runs the FDTD CAD in the background.

Launches FDTD CAD and stores a handle.

**run**(name, iter)

Saves simulation file and runs the simulation.

Copyright chriskeraly Copyright (c) 2019 Lumerical Inc.

lumopt.utilities.scipy_wrappers.trapz1D(vals, x)

This is trapz but that is ok with one dimensional values

lumopt.utilities.scipy_wrappers.trapz2D(vals, x, y)

2D trapz methods, which is OK with one dimensional values

lumopt.utilities.scipy_wrappers.trapz3D(vals, x, y, z)

3D trapz methods, which is OK with one dimensional values

lumopt.utilities.scipy_wrappers.wrapped_GridInterpolator(points, values, method='linear', bounds_error=True, fill_value=np)

This is a wrapper around Scipy’s RegularGridInterpolator so that it can deal with entries of 1 dimension

Original doc:

The data must be defined on a regular grid; the grid spacing however may be uneven. Linear and nearest-neighbour interpolation are supported. After setting up the interpolator object, the interpolation method *(linear or nearest)* may be chosen at each evaluation.

**points** [tuple of ndarray of float, with shapes (m1, ), ..., (mn, )] The points defining the regular grid in n dimensions.

**values** [array_like, shape (m1, ..., mn, ...)] The data on the regular grid in n dimensions.

**method** [str, optional] The method of interpolation to perform. Supported are “linear” and “nearest”. This parameter will become the default for the object’s __call__ method. Default is “linear”.

**bounds_error** [bool, optional] If True, when interpolated values are requested outside of the domain of the input data, a ValueError is raised. If False, then *fill_value* is used.
fill_value [number, optional] If provided, the value to use for points outside of the interpolation domain. If None, values outside the domain are extrapolated.

Copyright chriskeraly Copyright (c) 2019 Lumerical Inc.

lumopt.lumerical_methods.lumerical_scripts.get_fields(fdtd, monitor_name, get_eps, get_D, get_H, nointerpolation)

This function fetches fields from a monitor. For the fields used to create gradients, the D component of the field and the refractive index also need to be fetched. For the fields on figure of merit monitors, D or index does not need to be fetched It returns a Field object

lumopt.lumerical_methods.lumerical_scripts.get_fields_interp(fdtd, monitor_name, get_eps=False, get_D=False, get_H=False)

This function fetches fields from a monitor. For the fields used to create gradients, the D component of the field and the refractive index also need to be fetched. For the fields on figure of merit monitors, D or index does not need to be fetched It returns a Field object

lumopt.lumerical_methods.lumerical_scripts.get_fields_no_interp(fdtd, monitor_name, get_eps, get_D, get_H)

This function fetches fields from a monitor. For the fields used to create gradients, the D component of the field and the refractive index also need to be fetched. For the fields on figure of merit monitors, D or index does not need to be fetched It returns a Field object
CHAPTER 3

Indices and tables

- genindex
- modindex
- search
l

lumopt.lumerical_methods.lumerical_scripts, 17
lumopt.utilities.load_lumerical_scripts, 15
lumopt.utilities.scipy_wrappers, 16
A
AdaptiveGradientDescent (class in lumopt.optimizers.adaptive_gradient_descent), 14

B
boundary_perturbation_integrand() (lumopt.utilities.gradients.GradientFields method), 15

F
Fields (class in lumopt.utilities.fields), 15
FixedStepGradientDescent (class in lumopt.optimizers.fixed_step_gradient_descent), 14
FunctionDefinedPolygon (class in lumopt.geometries.polygon), 13

G
get_fields() (in module lumopt.lumerical_methods.lumerical_scripts), 17
get_fields_interp() (in module lumopt.lumerical_methods.lumerical_scripts), 17
get_fields_no_interp() (in module lumopt.lumerical_methods.lumerical_scripts), 17
GradientFields (class in lumopt.utilities.gradients), 15

L
load_from_lsf() (in module lumopt.utilities.load_lumerical_scripts), 15, 16
lumopt.lumerical_methods.lumerical_scripts (module), 17
lumopt.utilities.load_lumerical_scripts (module), 15, 16
lumopt.utilities.scipy_wrappers (module), 16

M
Material (class in lumopt.utilities.materials), 15
ModeMatch (class in lumopt.figures_of_merit.modematch), 12

P
Polygon (class in lumopt.geometries.polygon), 12
run() (lumopt.utilities.simulation.Simulation method), 16

S
ScipyOptimizers (class in lumopt.optimizers.generic_optimizers), 13
Simulation (class in lumopt.utilities.simulation), 16
spatial_gradient_integral() (lumopt.utilities.gradients.GradientFields method), 15

T
trapz1D() (in module lumopt.utilities.scipy_wrappers), 16
trapz2D() (in module lumopt.utilities.scipy_wrappers), 16
trapz3D() (in module lumopt.utilities.scipy_wrappers), 16

W
wrapped_GridInterpolator() (in module lumopt.utilities.scipy_wrappers), 16