LumOpt Documentation

Christopher Lalau-Keraly

Jul 05, 2019

Contents

CHAPTER 1

2D Crossing Optimization

CHAPTER 2

LumOpt's Documentation Table of Content

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** https://www.osapublishing.org/oe/abstract.cfm?uri=oe-21-18-21693
- **Shape optimization of nanophotonic devices using the adjoint method** http://ieeexplore.ieee.org/document/6989970/

Dissertations

- **Owen Miller (Berkeley, Yablonovitch group): Photonic Design: From Fundamental Solar Cell Physics to Computational I** http://optoelectronics.eecs.berkeley.edu/ThesisOwenMiller.pdf Focuses on the continuous adjoint method for electomagnetics, and has some very good intuitive explanations.
- **Paul Hansen (Stanford, Lambertus Hesselink group): Adjoint Sensitivity Analysis for NanoPhotonic Structures** http://purl.stanford.edu/zm205vy5828 Very thorough thesis on the mathematics, implementations and limitations of both Discrete and Continuous Adjoint methods
- **Samarth Bhargava (Berkeley, Yablonovitch group): Heat-Assisted Magnetic Recording: Fundamental Limits to Inverse E** http://optoelectronics.eecs.berkeley.edu/BhargavaDissertation.pdf Some neat applications of Continuous adjoint methods for HAMR (Heat Assisted Magnetic Recording) optical antenna design

- **Vidya Ganapati (Berkeley, Yablonovitch group): Optical Design Considerations for High Conversion Efficiency in Photov**
  http://optoelectronics.eecs.berkeley.edu/GanapatiDissertation.pdf Optimal surfaces for light concentration in Solar Cells
- **Christopher Lalau-Keraly (Berkeley, Yablonovitch group): Optimizing Nanophotonics: from Photoreceivers to Waveguid**
  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 3, with the latest version of Lumerical.

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*

```
""" 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)

(continued from previous page)

```
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 = 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 = np.linspace(0.25e-6, 0.6e-6, initial_points_x.size)
def taper_splitter(params = initial_points_y):
    ''' Defines a taper where the paramaters 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.interp1d(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 = 1e-9)

######## DEFINE FIGURE OF MERIT ########
# The base simulation script defines a field monitor named 'fom' at the point where␣
↪we want to modematch to the 3rd mode (fundamental TE mode).
```

(continues on next page)

(continued from previous page)

```
fom = ModeMatch(monitor_name = 'fom', mode_number = 2, direction = 'Forward', multi_
↪freq_src = True, target_T_fwd = lambda wl: np.ones(wl.size), norm_p = 1)

######## DEFINE OPTIMIZATION ALGORITHM ########
# This will run Scipy's implementation of the L-BFGS-B algoithm for at least 40␣
↪iterations. Since the variables are on the
# order of 1e-6, thery are scale up to be on the order of 1.
optimizer = ScipyOptimizers(max_iter = 30, method = 'L-BFGS-B', scaling_factor = 1e6,␣
↪pgtol = 1e-5)

######## PUT EVERYTHING TOGETHER ########
opt = Optimization(base_script = base_script, wavelengths = wavelengths, fom = fom,␣
↪geometry = geometry, optimizer = optimizer, hide_fdtd_cad = False, use_deps = True)

######## RUN THE OPTIMIZER ########
opt.run()
```

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:

```
# Copyright chriskeraly
# Copyright (c) 2019 Lumerical Inc.

switchtolayout;
selectall;
delete;

## SIM PARAMS
size_x=3e-6;
size_y=3e-6;
mesh_x=20e-9;
mesh_y=20e-9;
finer_mesh_size=2.5e-6;
mesh_accuracy=2;

## GEOMETRY
```

(continues on next page)

(continued from previous page)

```
#INPUT WAVEGUIDE
addrect;
set('name','input wg');
set('x span',3e-6);
set('y span',0.5e-6);
set('z span',220e-9);
set('y',0);
set('x',-2.5e-6);
set('index',2.8);

#OUTPUT WAVEGUIDES
addrect;
set('name','output wg top');
set('x span',3e-6);
set('y span',0.5e-6);
set('z span',220e-9);
set('y',0.35e-6);
set('x',2.5e-6);
set('index',2.8);

addrect;
set('name','output wg bottom');
set('x span',3e-6);
set('y span',0.5e-6);
set('z span',220e-9);
set('y',-0.35e-6);
set('x',2.5e-6);
set('index',2.8);

## SOURCE
addmode;
set('direction','Forward');
set('injection axis','x-axis');
#set('polarization angle',0);
set('y',0.0);
set('y span',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',1.44);
set('mesh accuracy',mesh_accuracy);
set('x',0.0);
set('x span',size_x);
set('y',0.0);
set('y span',size_y);
set('force symmetric y mesh',true);
set('y min bc','Anti-Symmetric');
set('pml layers',12);

## MESH IN OPTIMIZABLE REGION
addmesh;
set('x',0);
set('x span',finer_mesh_size+2.0*mesh_x);
```

(continues on next page)

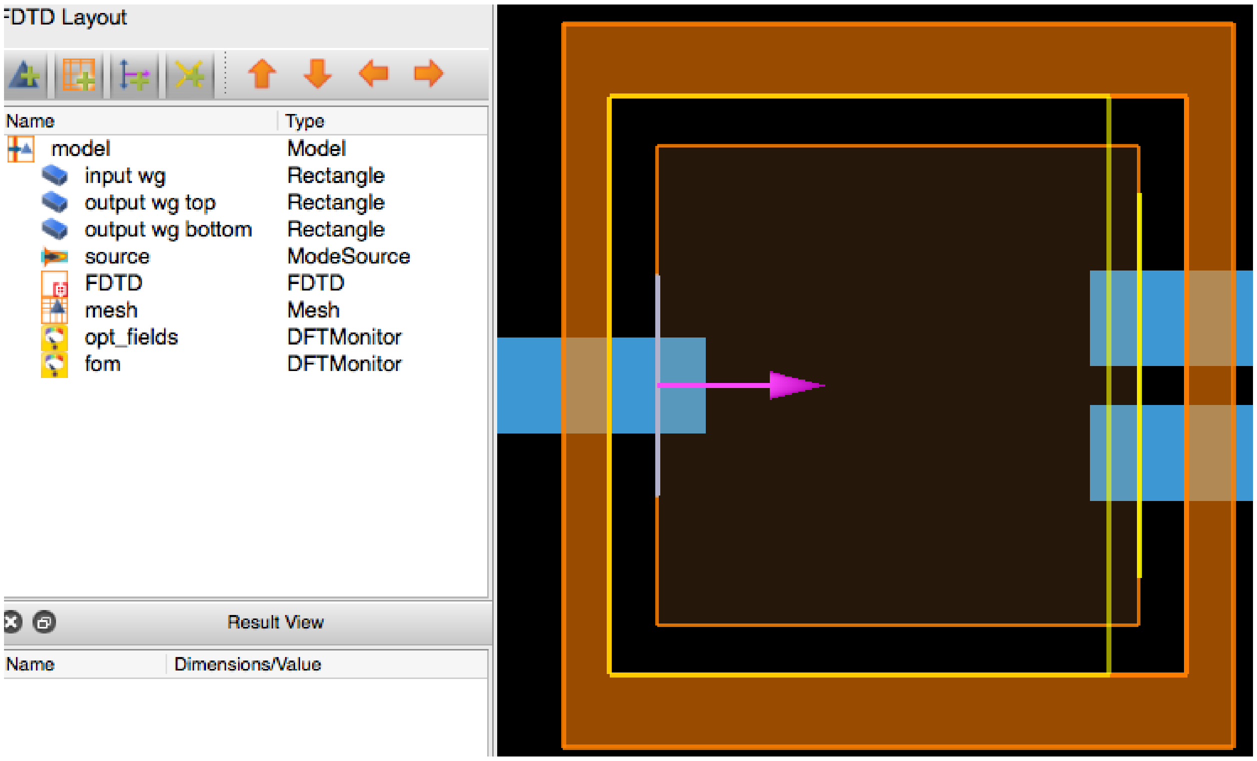
(continued from previous page)

```
set('y',0);
set('y span',finer_mesh_size);
set('dx',mesh_x);
set('dy',mesh_y);

## OPTIMIZATION FIELDS MONITOR IN OPTIMIZABLE REGION
addpower;
set('name','opt_fields');
set('monitor type','2D Z-normal');
set('x',0);
set('x span',finer_mesh_size);
set('y',0);
set('y span',finer_mesh_size);

## FOM FIELDS
addpower;
set('name','fom');
set('monitor type','2D X-normal');
set('x',finer_mesh_size/2.0);
set('y',0.0);
set('y span',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:



Name	Type
model	Model
input wg	Rectangle
output wg top	Rectangle
output wg bottom	Rectangle
source	ModeSource
FDTD	FDTD
mesh	Mesh
opt_fields	DFTMonitor
fom	DFTMonitor

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 succeptible 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 carefull 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 succeptible 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.

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

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

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 SPECTRAL RANGE #########
wavelengths = Wavelengths(start = 1550e-9, stop = 1550e-9, points = 1)

######## DEFINE BASE SIMULATION ########
# Use the same script for both simulations, but it's just to keep the example simple.␣
↪You could use two.
script_1 = os.path.join(os.path.dirname(__file__), 'splitter_base_TE_modematch.lsf')
script_2 = os.path.join(os.path.dirname(__file__), 'splitter_base_TE_modematch_
↪25nmoffset.lsf')

######## DEFINE OPTIMIZABLE GEOMETRY ########
## Here the two splitters just have a 25nm offset from each other, so that the result␣
↪is robust
```

(continues on next page)

(continued from previous page)

```
initial_points_x = np.linspace(-1.0e-6, 1.0e-6, 10)
initial_points_y = np.linspace(0.25e-6, 0.6e-6, initial_points_x.size)
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_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

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]))
    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_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

bounds = [(0.2e-6, 0.9e-6)] * initial_points_y.size
# guess from splitter_opt_2D.py optimization
initial_params = np.array([2.44788514e-07, 2.65915795e-07, 2.68748023e-07, 4.
↪42233947e-07, 6.61232152e-07, 6.47561406e-07, 6.91473099e-07, 6.17511522e-07, 6.
↪70669074e-07, 5.86141086e-07])
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
fom_1 = ModeMatch(monitor_name = 'fom', mode_number = 3, direction = 'Forward', multi_
↪freq_src = False, target_T_fwd = lambda wl: np.ones(wl.size), norm_p = 1)
fom_2 = ModeMatch(monitor_name = 'fom', mode_number = 3, direction = 'Forward', multi_
↪freq_src = False, target_T_fwd = lambda wl: np.ones(wl.size), norm_p = 1)

######## DEFINE OPTIMIZATION ALGORITHM ########
#For the optimizer, they should all be set the same, but different objects.␣
↪Eventually this will be improved
```

(continues on next page)

(continued from previous page)

```
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()
```

which yields:

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

class lumopt.optimization.**SuperOptimization**(*optimizations*)

Optimization super class to run two or more co-optimizations targeting different figures of merit that take the same parameters. The addition operator can be used to aggregate multiple optimizations. All the figures of merit are simply added to generate an overall figure of merit that is passed to the chosen optimizer.

Parameters **optimizations** – list of co-optimizations (each of class Optimization).

class lumopt.optimization.**Optimization**(*base_script, wavelengths, fom, geometry, optimizer, use_var_fdtd=False, hide_fdtd_cad=False, use_deps=True, plot_history=True, store_all_simulations=True*)

Acts as orchestrator for all the optimization pieces. Calling the member function run will perform the optimization, which requires four key pieces:

1) a script to generate the base simulation,
2) an object that defines and collects the figure of merit,
3) an object that generates the shape under optimization for a given set of optimization parameters and
4) a gradient based optimizer.

Parameters

- **`base_script`** – callable, file name or plain string with script to generate the base simulation.
- **`wavelengths`** – wavelength value (float) or range (class Wavelengths) with the spectral range for all simulations.
- **`fom`** – figure of merit (class ModeMatch).
- **`geometry`** – optimizable geometry (class FunctionDefinedPolygon).
- **`optimizer`** – SciyPy minimizer wrapper (class ScipyOptimizers).
- **`hide_fdtd_cad`** – flag run FDTD CAD in the background.
- **`use_deps`** – flag to use the numerical derivatives calculated directly from FDTD.
- **`plot_history`** – plot the history of all parameters (and gradients)
- **`store_all_simulations`** – Indicates if the project file for each iteration should be stored or not

2.5.2 Figures of Merit

class `lumopt.figures_of_merit.modematch.`**`ModeMatch`**(*monitor_name*, *mode_number*, *direction*, *multi_freq_src=False*, *target_T_fwd=<function ModeMatch.<lambda>>*, *norm_p=1*)

Calculates the figure of merit from an overlap integral between the fields recorded by a field monitor and the slected 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:

C. Lalau-Keraly, S. Bhargava, O. Miller, and E. Yablonovitch, "Adjoint shape optimization applied to electromagnetic design," Opt. Express 21, 21693-21701 (2013). https://doi.org/10.1364/OE.21.021693

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.
- **`multi_freq_src`** – bool flag to enable / disable a multi-frequency mode calculation and injection for the adjoint 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.

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

class `lumopt.optimizers.generic_optimizers.`**`ScipyOptimizers`**(*max_iter*, *method='L-BFGS-G'*, *scaling_factor=1.0*, *pgtol=1e-05*, *ftol=1e-12*, *target_fom=0*, *scale_initial_gradient_to=None*)

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 powerfull, 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 and gradient evaluations.
- **`method`** – string with the chosen minimization algorithm.
- **`scaling_factor`** – scalar or a vector of the same length as the optimization parameters; typically used to scale the optimization parameters so that they have magnitudes in the range zero to one.
- **`pgtol`** – projected gradient tolerance paramter 'gtol' (see 'BFGS' or 'L-BFGS-G' documentation).
- **`ftol`** – tolerance paramter 'ftol' which allows to stop optimization when changes in the FOM are less than this
- **`target_fom`** – A target value for the figure of merit. This allows to print/plot the distance of the current design from a target value
- **`scale_initial_gradient_to`** –

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 =

rac{ rac{dFOM}{dp_i}}{max_j(| rac{dFOM}{dp_j}|)}Delta x + ext{noise}_i

If all_params_equal = True, then the update equation is:

Delta p_i = sign(

rac{dFOM}{dp_i})Delta x + ext{noise}_i

If the optimization has many local optima: noise = rand([-1,1])*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 scalar or vector of the same length as the optimization parameters; typically used to scale the optimization parameters so that they have magnitudes in the range zero to one.

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*dx_regrowth_factor) while newfom < oldfom dx = dx / 2 if dx < min_dx:

dx = min_dx 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`** – scalar or vector of the same length as the optimization parameters; typically used to scale the optimization parameters so that they have magnitudes in the range zero to one.

2.5.5 Materials

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.

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

Loads the provided scritp 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

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.

`scale`(*dimension*, *factors*)

Scales the E, D and H field arrays along the specified dimension using the provided weighting factors.

Parameters

- **`dimension`** – 0 (x-axis), 1 (y-axis), 2 (z-axis), (3) frequency and (4) vector component.
- **`factors`** – list or vector of weighting factors of the same size as the target field dimension.

2.6.2 Gradient Fields

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.

`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.

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 scritp as a string and strips out all comments.

Parameters **`script_file_name`** – string specifying a file name.

class `lumopt.utilities.simulation.`**`Simulation`**(*workingDir*, *use_var_fdtd*, *hide_fdtd_cad*)

Object to manage the FDTD CAD.

Parameters

- **`workingDir`** – working directory for the CAD session.
- **`hide_fdtd_cad`** – 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.

class `lumopt.utilities.plotter.`**`Plotter`**(*movie=True*, *plot_history=True*)

Orchestrates the generation of plots during the optimization.

Parameters **`movie`** – Indicates if the evolution of parameters should be recorded as a movie

:param plot_history Indicates if we should plot the history of the parameters and gradients. Should be set to False for large (e.g. >100) numbers of parameters

class `lumopt.utilities.plotter.`**`SnapShots`**(**args*, *extra_args=None*, ***kwargs*)

Grabs the image information from the figure and saves it as a movie frame.

`finish`()

Finish any processing for writing the movie.

`grab_frame`(***fig_kwargs*)

All keyword arguments in fig_kwargs are passed on to the 'savefig' command that saves the figure.

`setup`(*fig*, *dpi*, *frame_prefix*)

Perform setup for writing the movie file.

fig [matplotlib.figure.Figure] The figure to grab the rendered frames from.

outfile [str] The filename of the resulting movie file.

dpi [number, optional] The dpi of the output file. This, with the figure size, controls the size in pixels of the resulting movie file. Default is fig.dpi.

frame_prefix [str, optional] The filename prefix to use for temporary files. Defaults to `'_tmp'`.

clear_temp [bool, optional] If the temporary files should be deleted after stitching the final result. Setting this to `False` can be useful for debugging. Defaults to `True`.

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

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.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index