EMUstack Documentation

Release 0.9.0

Björn Sturmberg

November 25, 2015

Contents

1 Introduction **1**
1.1 Introduction . . . 1

2 Installation **3**
2.1 Installation . . . 3

3 Guide **7**
3.1 Simulation Structure . . . 7
3.2 Single Interface . . . 7
3.3 Dispersion & Parallel Computation . . . 10
3.4 Thin Film Stack . . . 12
3.5 Including Metals . . . 13
3.6 1D Grating . . . 15
3.7 2D Grating . . . 17
3.8 Angles of Incidence & Eliptical Inclusions . . . 19
3.9 Plotting Fields 1D . . . 21
3.10 Plotting Fields 2D . . . 23
3.11 Plotting Amplitudes . . . 25
3.12 Shear Transformations . . . 27
3.13 Ultrathin Absorption Limit - Varying n . . . 29
3.14 Varying a Layer of a Stack . . . 31
3.15 Convergence Testing . . . 34
3.16 Extraordinary Optical Transmission . . . 36
3.17 Screen Sessions . . . 38

4 Python Backend **41**
4.1 objects module . . . 41
4.2 materials module . . . 45
4.3 mode_calcs module . . . 46
4.4 stack module . . . 48
4.5 plotting module . . . 49

5 Fortran Backends **57**
5.1 1D FEM Mode Solver . . . 57
5.2 2D FEM Mode Solver . . . 58

6 Indices and tables **59**

Python Module Index **61**

CHAPTER 1

Introduction

1.1 Introduction

EMUstack is an open-source simulation package for calculating light propagation through multi-layered stacks of dispersive, lossy, nanostructured, optical media. It implements a generalised scattering matrix method, which extends the physical intuition of thin film optics to complex structures.

At the heart of the scattering matrix approach is the requirement that each layer is uniform in one direction, here labelled z. In this nomenclature the incident field is unconstrained in $k_{\parallel} = k_{x,y}$ but must have $k_{\perp} = k_z \neq 0$.

In-plane each layer can be homogeneous, periodic in x or y, or double periodic (periodic in x and y). The modes of periodic (structured layers) are calculated using the Finite Element Method in respectively 1 or 2 dimensions, while the modes of homogeneous media are calculated analytically. This approach maximises the speed and accuracy of the calculations. These layers can be stacked in arbitrary order.

An advantage of EMUstack over other scattering matrix methods (for example CAMFR) is that the fields in each layer are considered in their natural basis with transmission scattering matrices converting fields between them. The fields in homogeneous layers are expressed in terms of plane waves, while the natural basis in the periodically structured layers are Bloch modes. Expressing fields in their natural basis gives the terms of the scattering matrices intuitive meaning, providing access to greater physical insights. It is also advantages for the speed and accuracy of the numerical method.

EMUstack has been designed to handle lossy media with dispersive refractive indices, with the complex refractive index at each frequency being taken directly from tabulated results of experimental measurements. This is an advantage of frequency domain methods over time domain methods such as the Finite Difference Time Domain (FDTD) where refractive indices are included by analytic approximations such as the Drude model. It is also possible to include media with lossless and/or non-dispersive refractive indices and EMUstack comes with a built in Drude model.

Taking full advantage of the boundary-element nature of the scattering matrix method it is possible to vary the thickness of a layer by a single, numerically inexpensive, matrix multiplication. Furthermore, EMUstack recognises when interfaces are repeated so that their scattering matrices need not be recalculated but rather just retrieved from memory, which takes practically no computation time.

EMUstack is a completely open source package, utilising free, open source compilers, meshing programs and libraries. All user interaction with EMUstack is done using the dynamic and easy to script language of python. The low-level numerical routines are written in Fortran for optimal performance making use of the LAPACK, ARPACK, and UMFPACK libraries. The Fortran routines are compiled as python subroutines using f2py. EMUstack currently comes with template FEM mesh for 1D and 2D gratings, Nanowire/Nanohole arrays, elliptical inclusions and split ring resonators. The mesh of other structures may be easily created using the open source program gmsh.

CHAPTER 2

Installation

2.1 Installation

The source code for EMUstack is hosted here on Github. Please download the latest release from here.

EMUstack has been developed on Ubuntu and is easiest to install on this platform. Simply 'sudo apt-get install' the packages listed in the dependencies.txt file and then run setup.sh.

```
$ sudo apt-get update
$ sudo apt-get -y install <dependencies>
$ /setup.sh
```

UPDATE: the current version of SuiteSparse is not fully compatible with 64 bit Linux... a solution to this is to backport SuiteSparse 3.4 from Ubuntu 12.04 using the method described here. Alternatively the pre-compiled libraries have been shown to work on Ubuntu 14.04

On other linux distributions either use the pre-compiled libraries of install them from the package manager or manually.

All that is required to use the pre-compiled libraries is to switch to a slightly modified Makefile and then run setup.sh.

```
$ cd backend/fortran/
$ mv Makefile Makefile_ubuntu
$ mv Makefile-pre_compiled_libs Makefile
$ cd ../../
$ /setup.sh
```

The Fortran components (EMUstack source code and libraries) have been successfully compiled with intel's ifortran as well as open-source gfortran. In this documentation we use gfortran.

NOTE: different versions of gmsh can give errors in the final test. This is okay, provided the test simulation ran, i.e. the test gives E rather than F.

2.1.1 SuiteSparse

The FEM routine used in EMUstack makes use of the highly optimised UMFPACK (Unsymmetric MultiFrontal Package) direct solver for sparse matrices developed by Prof. Timothy A. Davis. This is distributed as part of the SuiteSparse libraries under a GPL license. It can be downloaded from https://www.cise.ufl.edu/research/sparse/SuiteSparse/

This is the process I followed in my installations. They are provided as little more than tips...

Unpack SuiteSparse into EMUstack/backend/fortran/, it should create a directory there; SuiteSparse/ Make a directory where you want SuiteSparse installed, in my case SS_installed

```
$ mkdir SS_installed/
```

edit SuiteSparse/SuiteSparse_config/SuiteSparse_config.mk for consistency across the whole build; i.e. if using intel fortran compiler

```
line 75 F77 = gfortran --> ifort
```

set path to install folder:

```
line 85 INSTALL_LIB = /$Path_to_EMustack/EMUstack/backend/fortran/SS_install/lib
line 86 INSTALL_INCLUDE = /$Path_to_EMustack/EMUstack/backend/fortran/SS_install/include
```

line 290ish commenting out all other references to these:

```
F77 = ifort
CC = icc
BLAS   = -L/apps/intel-ct/12.1.9.293/mkl/lib/intel64 -lmkl_rt
LAPACK = -L/apps/intel-ct/12.1.9.293/mkl/lib/intel64 -lmkl_rt
```

Now make new directories for the paths you gave 2 steps back:

```
$ mkdir SS_installed/lib SS_installed/include
```

Download metis-4.0 and unpack metis into SuiteSparse/ Now move to the metis directory:

```
$ cd SuiteSparse/metis-4.0
```

Optionally edit metis-4.0/Makefile.in as per SuiteSparse/README.txt plus with -fPIC:

```
CC = gcc
or
CC = icc
OPTFLAGS = -O3 -fPIC
```

Now make metis (still in SuiteSparse/metis-4.0/):

```
$ make
```

Now move back to EMUstack/backend/fortran/

```
$ cp SuiteSparse/metis-4.0/libmetis.a SS_install/lib/
```

and then move to SuiteSparse/ and execute the following:

```
$ make library
$ make install
$ cd SuiteSparse/UMFPACK/Demo
$ make fortran64
$ cp SuiteSparse/UMFPACK/Demo/umf4_f77zwrapper64.o into SS_install/lib/
```

Copy the libraries into EMUstack/backend/fortran/Lib/ so that EMUstack/ is a complete package that can be moved across machine without alteration. This will override the pre-compiled libraries from the release (you may wish to save these somewhere).:

```
$ cp SS_install/lib/*.a EMUstack/backend/fortran/Lib/
$ cp SS_install/lib/umf4_f77zwrapper64.o EMUstack/backend/fortran/Lib/
```

2.1.2 EMUstack Makefile

Edit EMUstack/backend/fortran/Makefile to reflect what compiler you are using and how you installed the libraries. The Makefile has further details.

Then finally run the setup.sh script!

CHAPTER 3

Guide

3.1 Simulation Structure

Simulations with EMUstack are generally carried out using a python script file. This file is kept in its own directory which is placed in the EMUstack directory. All results of the simulation are automatically created within this directory. This directory then serves as a complete record of the calculation. Often, we will also save the simulation objects (scattering matrices, propagation constants etc.) within this folder for future inspection, manipulation, plotting, etc. Traditionally the name of the python script file begins with simo_. This is convenient for setting terminal alias' for running the script. Throughout the tutorial the script file will be called simo.py.

To start a simulation open a terminal and change into the directory containing the simo.py file. To run this script:

```
$ python simo.py
```

To have direct access to the simulation objects upon the completion of the script use,:

```
$ python -i simo.py
```

This will return you into an interactive python session in which all simulation objects are accessible. In this session you can access the docstrings of objects, classes and methods. For example:

```
>>> from pydoc import help
>>> help(objects.Light)
```

where we have accessed the docstring of the Light class from objects.py

In the remainder of the guide we go through a number of example simo.py files. These cover a wide range (though non-exhaustive) of established applications of EMUstack. The source files for these examples are in EMUstack/examples/ The first 8 examples are pretty essential for using EMUstack, while those thereafter show EMUstack applied to a number of (IMHO) interesting situations.

Another tip to mention before diving into the examples is running simulations within *screen_sesh*. These allow you to disconnect from the terminal instance and are discusses in *screen_sesh*.

3.2 Single Interface

```
"""
Simulating an interface between 2 homogeneous, non-dispersive media.
"""

import time
```

```
import datetime
import numpy as np
import sys
from multiprocessing import Pool
sys.path.append("../backend/")

import objects
import materials
import plotting
from stack import *

start = time.time()

################ Light parameters ####################
wl_1     = 500
wl_2     = 600
no_wl_1  = 4
# Set up light objects, starting with the wavelengths,
wavelengths = np.linspace(wl_1, wl_2, no_wl_1)
# and also specifying angles of incidence and refractive medium of semi-infinite
# layer that the light is incident upon (default value is n_inc = 1.0).
# Fields in homogeneous layers are expressed in a Fourier series of diffraction
# orders,where all orders within a radius of max_order_PWs in k-space are included.
light_list  = [objects.Light(wl, max_order_PWs = 1, theta = 0.0, phi = 0.0, \
    n_inc=1.5) for wl in wavelengths]

# Our structure must have a period, even if this is artificially imposed
# on a homogeneous thin film. What's more,
# it is critical that the period be consistent throughout a simulation!
period = 300

# Define each layer of the structure.
superstrate = objects.ThinFilm(period, height_nm = 'semi_inf',
    material = materials.Material(1.5 + 0.0j))
substrate   = objects.ThinFilm(period, height_nm = 'semi_inf',
    material = materials.Material(3.0 + 0.0j))

def simulate_stack(light):
    ################ Evaluate each layer individually ##############
    sim_superstrate = superstrate.calc_modes(light)
    sim_substrate   = substrate.calc_modes(light)
    ###################### Evaluate structure ######################
    """ Now define full structure. Here order is critical and
        stack list MUST be ordered from bottom to top!
    """

    stack = Stack((sim_substrate, sim_superstrate))
    # Calculate scattering matrices of the stack (for all polarisations).
    stack.calc_scat(pol = 'TE') # Incident light has TE polarisation,
    # which only effects the net transmission etc, not the matrices.

    return stack

stacks_list = map(simulate_stack, light_list)
# Save full simo data to .npz file for safe keeping!
np.savez('Simo_results', stacks_list=stacks_list)

# Calculation of the modes and scattering matrices of each layer
```

```
# as well as the scattering matrices of the interfaces of the stack
# is complete.
# From here on we can print, plot or manipulate the results.

# Alternatively, you may wish to finish the simo file here,
# and be output into an interactive python instance were you
# have access to all simulation objects and results for further
# manipulation. In this case you run this file as
# $ python -i simo_010-single_interface.py
# In this session the docstrings of objects/classes/methods
# can be accessed by typing

# >>> from pydoc import help
# >>> help(objects.Light)

# where we have accessed the docstring of the Light class from objects.py


######################## Post Processing ########################
# We can retrieve the propagation constants (k_z) of each layer.
# Let's print the values at the short wavelength in the superstrate,
wl_num = 0
lay = 1
betas = stacks_list[wl_num].layers[lay].k_z
print 'k_z of superstrate \n', betas
# and save the values for the longest wavelength for the substrate.
wl_num = -1
lay = 0
betas = stacks_list[wl_num].layers[lay].k_z
np.savetxt('Substrate_k_zs.txt', betas.view(float).reshape(-1, 2))
# Note that saving to txt files is slower than saving data as .npz
# However txt files may be easily read by other programs...


# We can also access the scattering matrices of individual layers,
# and of interfaces of the stack.
# For instance the reflection scattering matrix off the top
# of the substrate when considered as an isolated layer.
wl_num = -1
lay = 0
R12_sub = stacks_list[wl_num].layers[lay].R12
print 'R12 of substrate \n', R12_sub

# The reflection matrix for the reflection off the top of the
# superstrate-substrate interface meanwhile is a property of the stack.
R_interface = stacks_list[wl_num].R_net
# Let us plot this matrix in greyscale.
plotting.vis_scat_mats(R_interface)
# Since all layers are homogeneous this matrix should only have non-zero
# entries on the diagonal.

# Lastly, we can also plot the transmission, reflection, absorption
# of each layer and of the stack as a whole.
plotting.t_r_a_plots(stacks_list)

# p.s. we'll keep an eye on the time...
######################## Wrapping up ########################
print '\n*****************************************'
```

```
# Calculate and record the (real) time taken for simulation,
elapsed = (time.time() - start)
hms     = str(datetime.timedelta(seconds=elapsed))
hms_string = 'Total time for simulation was \n \
    %(hms)s (%(elapsed)12.3f seconds)'% {
            'hms'       : hms,
            'elapsed'   : elapsed, }
print hms_string
print '****************************************'
print ''

# and store this info.
python_log = open("python_log.log", "w")
python_log.write(hms_string)
python_log.close()
```

3.3 Dispersion & Parallel Computation

```
"""
Simulating an interface between 2 homogeneous, dispersive media.
We use multiple CPUs.
"""

import time
import datetime
import numpy as np
import sys
from multiprocessing import Pool
sys.path.append("../backend/")

import objects
import materials
import plotting
from stack import *

start = time.time()

# We begin by remove all results of previous simulations.
plotting.clear_previous()

################ Simulation parameters ################
# Select the number of CPUs to use in simulation.
num_cores = 2

################ Light parameters ####################
wl_1     = 400
wl_2     = 800
no_wl_1  = 4
# Set up light objects (no need to specifiy n_inc as light incident from
# Air with n_inc = 1.0).
wavelengths = np.linspace(wl_1, wl_2, no_wl_1)
light_list  = [objects.Light(wl, max_order_PWs = 1, theta = 0.0, phi = 0.0) \
    for wl in wavelengths]

# The period must be consistent throughout a simulation!
period = 300
```

```
# Define each layer of the structure, now with dispersive media.
# The refractive indices are interpolated from tabulated data.
superstrate = objects.ThinFilm(period, height_nm = 'semi_inf',
    material = materials.Air)
substrate   = objects.ThinFilm(period, height_nm = 'semi_inf',
    material = materials.SiO2_a) # Amorphous silica

def simulate_stack(light):
    ################ Evaluate each layer individually ##############
    sim_superstrate = superstrate.calc_modes(light)
    sim_substrate   = substrate.calc_modes(light)
    ###################### Evaluate structure ######################
    """ Now define full structure. Here order is critical and
        stack list MUST be ordered from bottom to top!
    """

    stack = Stack((sim_substrate, sim_superstrate))
    stack.calc_scat(pol = 'TM') # This time TM polarised light is incident.

    return stack

# Run wavelengths in parallel across num_cores CPUs using multiprocessing package.
pool = Pool(num_cores)
stacks_list = pool.map(simulate_stack, light_list)
# Save full simo data to .npz file for safe keeping!
np.savez('Simo_results', stacks_list=stacks_list)

######################## Post Processing ########################
# This time let's visualise the net Transmission scattering matrix,
# which describes the propagation of light all the way from the superstrate into
# the substrate. When studying diffractive layers it is useful to know how many
# of theplane waves of the substrate are propagating, so lets include this.
wl_num = -1
T_net = stacks_list[wl_num].T_net
nu_prop = stacks_list[wl_num].layers[0].num_prop_pw_per_pol
plotting.vis_scat_mats(T_net, nu_prop_PWs=nu_prop)

# Let's just plot the spectra and see the effect of changing refractive indices.
plotting.t_r_a_plots(stacks_list)

######################## Wrapping up ########################
print '\n*****************************************'
# Calculate and record the (real) time taken for simulation,
elapsed = (time.time() - start)
hms     = str(datetime.timedelta(seconds=elapsed))
hms_string = 'Total time for simulation was \n \
    %(hms)s (%(elapsed)12.3f seconds)'% {
            'hms'       : hms,
            'elapsed'   : elapsed, }
print hms_string
print '*****************************************'
print ''

# and store this info.
python_log = open("python_log.log", "w")
python_log.write(hms_string)
python_log.close()
```

3.4 Thin Film Stack

```
"""
Simulating a stack of homogeneous, dispersive media.
"""

import time
import datetime
import numpy as np
import sys
from multiprocessing import Pool
sys.path.append("../backend/")

import objects
import materials
import plotting
from stack import *

start = time.time()

# Remove results of previous simulations.
plotting.clear_previous()

################ Simulation parameters ################
# Select the number of CPUs to use in simulation.
num_cores = 2

################ Light parameters ####################
wl_1     = 400
wl_2     = 800
no_wl_1  = 4
wavelengths = np.linspace(wl_1, wl_2, no_wl_1)
light_list  = [objects.Light(wl, max_order_PWs = 1, theta = 0.0, phi = 0.0) \
for wl in wavelengths]

# The period must be consistent throughout a simulation!
period = 300

# Define each layer of the structure.
superstrate = objects.ThinFilm(period, height_nm = 'semi_inf',
    material = materials.Air)
# Define a thin film with (finite) thickness in nm and constant refractive index
TF_1 = objects.ThinFilm(period, height_nm = 100,
    material = materials.Material(1.0 + 0.05j))
# EMUstack calculation time is independent dispersion and thickness of layer!
# This layer is made of Indium Phosphide, the tabulated refractive index of which
# is stored in EMUstack/data/
# We artificially set the imaginary part of the layer to zero for all wavelengths.
TF_2 = objects.ThinFilm(period, height_nm = 5e6,
    material = materials.InP, loss=False)
# By default loss = True
TF_3 = objects.ThinFilm(period, height_nm = 52,
    material = materials.Si_a)
# Note that the semi-inf substrate must be lossess so that EMUstack can distinguish
# propagating plane waves that carry energy from evanescent waves which do not.
# This layer is therefore crystalline silicon with Im(n) == 0.
substrate   = objects.ThinFilm(period, height_nm = 'semi_inf',
    material = materials.Si_c, loss=False)
```

```
def simulate_stack(light):
    ############### Evaluate each layer individually #############
    sim_superstrate = superstrate.calc_modes(light)
    sim_TF_1 = TF_1.calc_modes(light)
    sim_TF_2 = TF_2.calc_modes(light)
    sim_TF_3 = TF_3.calc_modes(light)
    sim_substrate   = substrate.calc_modes(light)
    ###################### Evaluate structure ####################
    """ Now define full structure. Here order is critical and
        stack list MUST be ordered from bottom to top!
    """
# We can now stack these layers of finite thickness however we wish.
    stack = Stack((sim_substrate, sim_TF_1, sim_TF_3, sim_TF_2, sim_TF_1, \
        sim_superstrate))
    stack.calc_scat(pol = 'TM')

    return stack

# Run wavelengths in parallel across num_cores CPUs using multiprocessing package.
pool = Pool(num_cores)
stacks_list = pool.map(simulate_stack, light_list)
# Save full simo data to .npz file for safe keeping!
np.savez('Simo_results', stacks_list=stacks_list)

######################## Post Processing ########################
# We will now see the absorption in each individual layer as well as of the stack.
plotting.t_r_a_plots(stacks_list)

######################## Wrapping up ########################
print '\n########################################'
# Calculate and record the (real) time taken for simulation,
elapsed = (time.time() - start)
hms     = str(datetime.timedelta(seconds=elapsed))
hms_string = 'Total time for simulation was \n \
    %(hms)s (%(elapsed)12.3f seconds)'% {
            'hms'       : hms,
            'elapsed'   : elapsed, }
print hms_string
print '########################################'
print ''

# and store this info.
python_log = open("python_log.log", "w")
python_log.write(hms_string)
python_log.close()
```

3.5 Including Metals

```
"""
EUMstack loves metal \m/
However, as we saw in the previous example the substrate layer must be lossless,
so that we can distinguish propagating waves from evanescent ones.
To terminate the stack with a metalic mirror we must make it finite, but very thick.
"""
```

```
import time
import datetime
import numpy as np
import sys
from multiprocessing import Pool
sys.path.append("../backend/")

import objects
import materials
import plotting
from stack import *

start = time.time()

# Remove results of previous simulations.
plotting.clear_previous()

################ Simulation parameters ################
# Select the number of CPUs to use in simulation.
num_cores = 2

################ Light parameters ####################
wl_1     = 400
wl_2     = 800
no_wl_1  = 4
wavelengths = np.linspace(wl_1, wl_2, no_wl_1)
light_list  = [objects.Light(wl, max_order_PWs = 1, theta = 0.0, phi = 0.0)\
    for wl in wavelengths]

# The period must be consistent throughout a simulation!
period = 300

# Define each layer of the structure, as in last example.
superstrate = objects.ThinFilm(period, height_nm = 'semi_inf',
    material = materials.Air)
TF_2 = objects.ThinFilm(period, height_nm = 5e6,
    material = materials.InP, loss=False)
TF_3 = objects.ThinFilm(period, height_nm = 52,
    material = materials.Si_a)
# Realistically a few micron thick mirror would do the trick,
# but EMUstack is height agnostic.... so what the hell.
mirror = objects.ThinFilm(period, height_nm = 1e5,
    material = materials.Ag)
substrate   = objects.ThinFilm(period, height_nm = 'semi_inf',
    material = materials.Air)

def simulate_stack(light):
    ############### Evaluate each layer individually ##############
    sim_superstrate = superstrate.calc_modes(light)
    sim_mirror = mirror.calc_modes(light)
    sim_TF_2 = TF_2.calc_modes(light)
    sim_TF_3 = TF_3.calc_modes(light)
    sim_substrate   = substrate.calc_modes(light)
    ###################### Evaluate structure #####################
    """ Now define full structure. Here order is critical and
        stack list MUST be ordered from bottom to top!
    """
# Put semi-inf substrate below thick mirror so that propagating energy is defined.
```

```
    stack = Stack((sim_substrate, sim_mirror, sim_TF_3, sim_TF_2, sim_superstrate))
    stack.calc_scat(pol = 'TM')

    return stack

pool = Pool(num_cores)
stacks_list = pool.map(simulate_stack, light_list)
# Save full simo data to .npz file for safe keeping!
np.savez('Simo_results', stacks_list=stacks_list)

######################## Post Processing ########################
# The total transmission should be zero.
plotting.t_r_a_plots(stacks_list)

######################## Wrapping up ########################
print '\n#########################################'
# Calculate and record the (real) time taken for simulation,
elapsed = (time.time() - start)
hms     = str(datetime.timedelta(seconds=elapsed))
hms_string = 'Total time for simulation was \n \
    %(hms)s (%(elapsed)12.3f seconds)'% {
            'hms'       : hms,
            'elapsed'   : elapsed, }
print hms_string
print '#########################################'
print ''

# and store this info.
python_log = open("python_log.log", "w")
python_log.write(hms_string)
python_log.close()
```

3.6 1D Grating

```
"""
Simulating a lamellar grating that is periodic in x only.
For this simulation EMUstack uses the 1D diffraction orders for the basis
of the plane waves and carries out a 1D FEM calculation for the modes of
the grating.
"""

import time
import datetime
import numpy as np
import sys
from multiprocessing import Pool
sys.path.append("../backend/")

import objects
import materials
import plotting
from stack import *

start = time.time()

# Remove results of previous simulations.
```

```
plotting.clear_previous()

################ Simulation parameters ################
# Select the number of CPUs to use in simulation.
num_cores = 2

################ Light parameters #####################
wl_1     = 400
wl_2     = 800
no_wl_1  = 2
wavelengths = np.linspace(wl_1, wl_2, no_wl_1)
light_list  = [objects.Light(wl, max_order_PWs = 5, theta = 0.0, phi = 0.0) for wl in wavelengths]

# The period must be consistent throughout a simulation!
period = 300

# Define each layer of the structure
# We need to inform EMUstack at this point that all layers in the stack will
# be at most be periodic in one dimension (i.e. there are no '2D_arrays's).
# This is done with the Keyword Arg 'world_1d' and all homogenous layers are
# calculated using the PW basis of 1D diffraction orders.
superstrate = objects.ThinFilm(period, height_nm = 'semi_inf', world_1d=True,
    material = materials.Air)

substrate   = objects.ThinFilm(period, height_nm = 'semi_inf', world_1d=True,
    material = materials.Air)
# Define 1D grating that is periodic in x.
# The mesh for this is always made 'live' in objects.py the number of
# FEM elements used is given by 1/lc_bkg.
# See Fortran Backends section of tutorial for more details.
grating = objects.NanoStruct('1D_array', period, int(round(0.75*period)), height_nm = 2900,
    background = materials.Material(1.46 + 0.0j), inclusion_a = materials.Material(5.0 + 0.0j),
    loss = True, lc_bkg = 0.0051)

def simulate_stack(light):
    ################ Evaluate each layer individually ##############
    sim_superstrate = superstrate.calc_modes(light)
    sim_grating     = grating.calc_modes(light)
    sim_substrate   = substrate.calc_modes(light)
    ###################### Evaluate structure ######################
    """ Now define full structure. Here order is critical and
        stack list MUST be ordered from bottom to top!
    """

    stack = Stack((sim_substrate, sim_grating, sim_superstrate))
    stack.calc_scat(pol = 'TE')

    return stack

pool = Pool(num_cores)
# stacks_list = pool.map(simulate_stack, light_list)
stacks_list = map(simulate_stack, light_list)
# Save full simo data to .npz file for safe keeping!
np.savez('Simo_results', stacks_list=stacks_list)

######################## Post Processing ########################
# The total transmission should be zero.
plotting.t_r_a_plots(stacks_list)
```

```
######################## Wrapping up ########################
print '\n*****************************************'
# Calculate and record the (real) time taken for simulation,
elapsed = (time.time() - start)
hms     = str(datetime.timedelta(seconds=elapsed))
hms_string = 'Total time for simulation was \n \
    %(hms)s (%(elapsed)12.3f seconds)'% {
            'hms'       : hms,
            'elapsed'   : elapsed, }
print hms_string
print '*****************************************'
print ''

# and store this info.
python_log = open("python_log.log", "w")
python_log.write(hms_string)
python_log.close()
```

3.7 2D Grating

```
"""
Simulating a nanowire array with period 600 nm and NW diameter 120 nm.
"""

import time
import datetime
import numpy as np
import sys
from multiprocessing import Pool
sys.path.append("../backend/")

import objects
import materials
import plotting
from stack import *

start = time.time()
################ Simulation parameters ################

# Number of CPUs to use in simulation
num_cores = 7

# Remove results of previous simulations
plotting.clear_previous()

################ Light parameters ####################
wl_1     = 310
wl_2     = 1127
no_wl_1  = 3
# Set up light objects
wavelengths = np.linspace(wl_1, wl_2, no_wl_1)
light_list  = [objects.Light(wl, max_order_PWs = 2, theta = 0.0, phi = 0.0) \
    for wl in wavelengths]

# Period must be consistent throughout simulation!!!
```

```
period = 600

# In this example we set the number of Bloch modes to use in the simulation
# Be default it is set to be slightly greater than the number of PWs.
num_BMs = 200

superstrate = objects.ThinFilm(period, height_nm = 'semi_inf',
    material = materials.Air, loss = False)

substrate  = objects.ThinFilm(period, height_nm = 'semi_inf',
    material = materials.SiO2, loss = False)

NW_diameter = 120
NW_array = objects.NanoStruct('2D_array', period, NW_diameter, height_nm = 2330,
    inclusion_a = materials.Si_c, background = materials.Air, loss = True,
    make_mesh_now = True, force_mesh = True, lc_bkg = 0.1, lc2= 2.0)
# Here we get EMUstack to make the FEM mesh automagically using our input parameters.
# the lc_bkg parameter sets the baseline distance between points on the FEM mesh,
# lc_bkg/lc2 is the distance between mesh points that lie on the inclusion boundary.
# There are higher lc parameters which are used when including multiple inclusions.

# Alternatively we can specify a pre-made mesh as follows.
NW_array2 = objects.NanoStruct('2D_array', period, NW_diameter, height_nm = 2330,
    inclusion_a = materials.Si_c, background = materials.Air, loss = True,
    make_mesh_now = False, mesh_file='4testing-600_120.mail')


def simulate_stack(light):

    ################ Evaluate each layer individually ##############
    sim_superstrate = superstrate.calc_modes(light)
    sim_substrate   = substrate.calc_modes(light)
    sim_NWs         = NW_array.calc_modes(light, num_BMs=num_BMs)

    ###################### Evaluate structure ######################
    """ Now define full structure. Here order is critical and
        stack list MUST be ordered from bottom to top!
    """

    stack = Stack((sim_substrate, sim_NWs, sim_superstrate))
    stack.calc_scat(pol = 'TE')

    return stack


# Run in parallel across wavelengths.
pool = Pool(num_cores)
stacks_list = pool.map(simulate_stack, light_list)
# Save full simo data to .npz file for safe keeping!
np.savez('Simo_results', stacks_list=stacks_list)


######################## Plotting ########################

# We here wish to know the photovoltaic performance of the structure,
# where all light absorbed in the NW layer is considered to produce exactly
# one electron-hole pair.
# To do this we specify which layer of the stack is the PV active layer
```

```
# (default active_layer_nu=1), and indicate that we want to calculate
# the ideal short circuit current (J_sc) of the cell.
# We could also calculate the 'ultimate efficiency' by setting ult_eta=True.
plotting.t_r_a_plots(stacks_list, active_layer_nu=1, J_sc=True)

# We also plot the dispersion relation for each layer.
plotting.omega_plot(stacks_list, wavelengths)

######################## Wrapping up ########################
# Calculate and record the (real) time taken for simulation
elapsed = (time.time() - start)
hms     = str(datetime.timedelta(seconds=elapsed))
hms_string = 'Total time for simulation was \n \
    %(hms)s (%(elapsed)12.3f seconds)'% {
            'hms'       : hms,
            'elapsed'   : elapsed, }

python_log = open("python_log.log", "w")
python_log.write(hms_string)
python_log.close()

print hms_string
print '*****************************************'
print ''
```

3.8 Angles of Incidence & Eliptical Inclusions

```
"""
Simulating circular dichroism effect in elliptic nano hole arrays
as in T Cao1 and Martin J Cryan doi:10.1088/2040-8978/14/8/085101.
"""

import time
import datetime
import numpy as np
import sys
from multiprocessing import Pool
sys.path.append("../backend/")

import objects
import materials
import plotting
from stack import *

start = time.time()
################ Simulation parameters ################

# Number of CPUs to use in simulation
num_cores = 4

# Remove results of previous simulations
plotting.clear_previous()

################ Light parameters ####################
wl_1     = 300
wl_2     = 1000
```

```
no_wl_1  = 21
# Set up light objects
wavelengths = np.linspace(wl_1, wl_2, no_wl_1)
light_list  = [objects.Light(wl, theta = 45, phi = 45, max_order_PWs = 2) \
    for wl in wavelengths]



# Period must be consistent throughout simulation!!!
period = 165
diam1  = 140
diam2  = 60
ellipticity = (float(diam1-diam2))/float(diam1)

# Replicating the geometry of the paper we set up a gold layer with elliptical air
# holes. To get good agreement with the published work we use the Drude model for Au.
# Note that better physical results are obtained using the tabulated data for Au!
Au_NHs = objects.NanoStruct('2D_array', period, diam1, inc_shape = 'ellipse',
    ellipticity = ellipticity, height_nm = 60,
    inclusion_a = materials.Air, background = materials.Au_drude, loss = True,
    make_mesh_now = True, force_mesh = True, lc_bkg = 0.2, lc2= 5.0)

superstrate = objects.ThinFilm(period = period, height_nm = 'semi_inf',
    material = materials.Air, loss = True)
substrate = objects.ThinFilm(period = period, height_nm = 'semi_inf',
    material = materials.Air, loss = False)

# Again for this example we fix the number of BMs.
num_BMs = 50

def simulate_stack(light):
    ################ Evaluate each layer individually ##############
    sim_superstrate = superstrate.calc_modes(light)
    sim_Au   = Au_NHs.calc_modes(light, num_BMs = num_BMs)
    sim_substrate = substrate.calc_modes(light)

    stackSub = Stack((sim_substrate, sim_Au, sim_superstrate))
    stackSub.calc_scat(pol = 'R Circ')
    stackSub2 = Stack((sim_substrate, sim_Au, sim_superstrate))
    stackSub2.calc_scat(pol = 'L Circ')
    saveStack = Stack((sim_substrate, sim_Au, sim_superstrate))

    a_CD = []
    t_CD = []
    r_CD = []
    for i in range(len(stackSub.a_list)):
        a_CD.append(stackSub.a_list.pop() - stackSub2.a_list.pop())
    for i in range(len(stackSub.t_list)):
        t_CD.append(stackSub.t_list.pop() - stackSub2.t_list.pop())
    for i in range(len(stackSub.r_list)):
        r_CD.append(stackSub.r_list.pop() - stackSub2.r_list.pop())
    saveStack.a_list = a_CD
    saveStack.t_list = t_CD
    saveStack.r_list = r_CD

    return saveStack
```

```
# Run in parallel across wavelengths.
pool = Pool(num_cores)
stacks_list = pool.map(simulate_stack, light_list)
# Save full simo data to .npz file for safe keeping!
np.savez('Simo_results', stacks_list=stacks_list)

######################## Plotting ########################
# Just to show how it's done we can add the height of the layer and some extra
# details to the file names and plot titles.
title = 'what_a_lovely_day-'

plotting.t_r_a_plots(stacks_list, add_height=Au_NHs.height_nm, add_name=title)


# Calculate and record the (real) time taken for simulation
elapsed = (time.time() - start)
hms     = str(datetime.timedelta(seconds=elapsed))
hms_string = 'Total time for simulation was \n \
    %(hms)s (%(elapsed)12.3f seconds)'% {
            'hms'       : hms,
            'elapsed'   : elapsed, }

python_log = open("python_log.log", "w")
python_log.write(hms_string)
python_log.close()

print '****************************************'
print hms_string
print '****************************************'
print ''
```

3.9 Plotting Fields 1D

```
"""
Show how to plot electric fields.
"""

import time
import datetime
import numpy as np
import sys
from multiprocessing import Pool
sys.path.append("../backend/")

import objects
import materials
import plotting
from stack import *

start = time.time()
################ Simulation parameters ################

# Number of CPUs to use in simulation
num_cores = 7

# Remove results of previous simulations
```

```
plotting.clear_previous()

################ Light parameters ####################
wl     = 615
light_list  = [objects.Light(wl, max_order_PWs = 10, theta = 0.0, phi = 0.0)]

# Period must be consistent throughout simulation!!!
period = 600

superstrate = objects.ThinFilm(period, height_nm = 'semi_inf', world_1d = True,
    material = materials.Air, loss = False)

substrate  = objects.ThinFilm(period, height_nm = 'semi_inf', world_1d = True,
    material = materials.Air, loss = False)

spacer  = objects.ThinFilm(period, height_nm = 200, world_1d = True,
    material = materials.SiO2_a, loss = True)

grating = objects.NanoStruct('1D_array', period, int(round(0.7*period)), height_nm = 400,
    background = materials.Material(1.45 + 0.0j),
    inclusion_a = materials.Material(3.77 + 0.01j),
    loss = True, lc_bkg = 0.005, plotting_fields = True)


def simulate_stack(light):

    ################ Evaluate each layer individually ##############
    sim_superstrate = superstrate.calc_modes(light)
    sim_substrate   = substrate.calc_modes(light)
    sim_grating     = grating.calc_modes(light)
    sim_spacer      = spacer.calc_modes(light)

    ###################### Evaluate structure ######################
    """ Now define full structure. Here order is critical and
        stack list MUST be ordered from bottom to top!
    """

    stack = Stack((sim_substrate, sim_spacer, sim_grating, sim_superstrate))
    stack.calc_scat(pol = 'TE')

    return stack


# Run in parallel across wavelengths.
pool = Pool(num_cores)
stacks_list = pool.map(simulate_stack, light_list)
# Save full simo data to .npz file for safe keeping!
np.savez('Simo_results', stacks_list = stacks_list)


######################## Plotting ########################

# Plot fields on slices through stack.
#
# Note that all field plots of previous simulations are deleted! Move any
# results that you wish to keep into a different folder, ideally copying the
# whole simo directory to future reference to simo parameters.
#
```

```
# plotting.fields_vertically(stacks_list)
# # We can also plot only the scattered field (disregarding the incident field)
# plotting.fields_vertically(stacks_list, no_incoming = True, add_name = '-no_incoming')
#
# The above fields are the total fields, we can also look at the fields of
# each individual Bloch mode, which for a 1D array is done like so,
plotting.Bloch_fields_1d(stacks_list)

######################## Wrapping up ########################
# Calculate and record the (real) time taken for simulation
elapsed = (time.time() - start)
hms     = str(datetime.timedelta(seconds=elapsed))
hms_string = 'Total time for simulation was \n \
    %(hms)s (%(elapsed)12.3f seconds)'% {
            'hms'       : hms,
            'elapsed'   : elapsed, }

python_log = open("python_log.log", "w")
python_log.write(hms_string)
python_log.close()

print hms_string
print '*****************************************'
print ''
```

3.10 Plotting Fields 2D

```
"""
Show how to plot electric fields.
"""

import time
import datetime
import numpy as np
import sys
from multiprocessing import Pool
sys.path.append("../backend/")

import objects
import materials
import plotting
from stack import *

start = time.time()
################ Simulation parameters ################

# Number of CPUs to use in simulation
num_cores = 7

# Remove results of previous simulations
plotting.clear_previous()

################ Light parameters ####################
wl     = 615
light_list  = [objects.Light(wl, max_order_PWs = 15, theta = 0.0, phi = 0.0)]
```

```
# Period must be consistent throughout simulation!!!
period = 600

superstrate = objects.ThinFilm(period, height_nm = 'semi_inf',
    material = materials.Air, loss = False)

substrate  = objects.ThinFilm(period, height_nm = 'semi_inf',
    material = materials.Air, loss = False)

spacer  = objects.ThinFilm(period, height_nm = 200,
    material = materials.SiO2_a, loss = True)

NW_diameter = 120
NW_array = objects.NanoStruct('2D_array', period, NW_diameter,
    height_nm = 2330, inclusion_a = materials.Si_c, background = materials.Air,
    loss = True, make_mesh_now = True, force_mesh = True, lc_bkg = 0.1,
    lc2= 2.0, plotting_fields = True)


def simulate_stack(light):

    ################ Evaluate each layer individually ##############
    sim_superstrate = superstrate.calc_modes(light)
    sim_substrate   = substrate.calc_modes(light)
    sim_NWs         = NW_array.calc_modes(light)
    sim_spacer      = spacer.calc_modes(light)

    ###################### Evaluate structure ######################
    """ Now define full structure. Here order is critical and
        stack list MUST be ordered from bottom to top!
    """

    stack = Stack((sim_substrate, sim_spacer, sim_NWs, sim_superstrate))
    stack.calc_scat(pol = 'TE')

    return stack


# Run in parallel across wavelengths.
pool = Pool(num_cores)
stacks_list = pool.map(simulate_stack, light_list)
# Save full simo data to .npz file for safe keeping!
np.savez('Simo_results', stacks_list = stacks_list)


######################## Plotting ########################

# Plot fields on slices through stack along the x & y axis,
# and along the diagonals.
# This is done through all layers of the stack and saved as png files.
#
# Note that all field plots of previous simulations are deleted! Move any
# results that you wish to keep into a different folder, ideally copying the
# whole simo directory to future reference to simo parameters.
#
plotting.fields_vertically(stacks_list)

# Plot fields in the x-y plane at a list of specified heights.
```

```
plotting.fields_in_plane(stacks_list, lay_interest = 2, z_values = [0.0, 2.0])
plotting.fields_in_plane(stacks_list, lay_interest = 1, z_values = [1.0, 3.2])

# Plot fields inside nanostructures in 3D which are viewed using gmsh.
plotting.fields_3d(stacks_list, lay_interest = 2)

# Save electric field values (all components) at a list of selected point.
plotting.field_values(stacks_list, lay_interest = 0, xyz_values = [(4.0, 2.5, 7.0), (1.0, 1.5, 3.0)])

######################### Wrapping up ########################
# Calculate and record the (real) time taken for simulation
elapsed = (time.time() - start)
hms     = str(datetime.timedelta(seconds=elapsed))
hms_string = 'Total time for simulation was \n \
    %(hms)s (%(elapsed)12.3f seconds)'% {
            'hms'       : hms,
            'elapsed'   : elapsed, }

python_log = open("python_log.log", "w")
python_log.write(hms_string)
python_log.close()

print hms_string
print '*****************************************'
print ''
```

3.11 Plotting Amplitudes

```
"""
Here we investigate how efficiently a stack of 1D gratings excite diffraction orders.
"""

import time
import datetime
import numpy as np
import sys
from multiprocessing import Pool
sys.path.append("../backend/")

import objects
import materials
import plotting
from stack import *

start = time.time()
################ Simulation parameters ################

# Number of CPUs to use in simulation
num_cores = 5

# Remove results of previous simulations
plotting.clear_previous()
################ Light parameters ####################
wavelengths = np.linspace(1500,1600,10)
light_list  = [objects.Light(wl, max_order_PWs = 6, theta = 0.0, phi = 0.0) \
    for wl in wavelengths]
```

```
################ Grating parameters ####################
# The period must be consistent throughout a simulation!
period = 700

superstrate = objects.ThinFilm(period, height_nm = 'semi_inf', world_1d = True,
    material = materials.Air, loss = False)

substrate  = objects.ThinFilm(period, height_nm = 'semi_inf', world_1d = True,
    material = materials.Air, loss = False)

absorber    = objects.ThinFilm(period, height_nm = 10, world_1d = True,
    material = materials.Material(1.0 + 0.05j), loss = True)

grating_1 = objects.NanoStruct('1D_array', period, int(round(0.75*period)),
    height_nm = 2900, background = materials.Material(1.46 + 0.0j),
    inclusion_a = materials.Material(3.61 + 0.0j), loss = True,
    lc_bkg = 0.005)


def simulate_stack(light):

    ################ Evaluate each layer individually ##############
    sim_superstrate = superstrate.calc_modes(light)
    sim_substrate   = substrate.calc_modes(light)
    sim_absorber    = absorber.calc_modes(light)
    sim_grating_1   = grating_1.calc_modes(light)

    ###################### Evaluate structure ######################
    """ Now define full structure. Here order is critical and
        stack list MUST be ordered from bottom to top!
    """

    stack = Stack((sim_substrate, sim_absorber, sim_grating_1, sim_superstrate))
    stack.calc_scat(pol = 'TE')

    return stack


# Run in parallel across wavelengths.
pool = Pool(num_cores)
stacks_list = pool.map(simulate_stack, light_list)
# Save full simo data to .npz file for safe keeping!
np.savez('Simo_results', stacks_list = stacks_list)

######################## Post Processing ########################
# We can plot the amplitudes of each transmitted plane wave order as a
# function of angle.
plotting.PW_amplitudes(stacks_list, add_name = '-default_substrate')
# By default this will plot the amplitudes in the substrate, however we can also give
# the index in the stack of a different homogeneous layer and calculate them here.
# We here chose a subset of orders to plot.
plotting.PW_amplitudes(stacks_list, chosen_PWs = [-1,0,2], \
    lay_interest = 1)

# When many plane wave orders are included these last plots can become confusing,
# so instead one may wish to sum together the amplitudes of all propagating orders,
# of all evanescent orders, and all far-evanescent orders
```

```
# (which have in plane k>n_H * k0).
plotting.evanescent_merit(stacks_list, lay_interest = 0)


plotting.BM_amplitudes(stacks_list, lay_interest = 2, chosen_BMs = [0,1,2,3,4,5])

# Lastly we also plot the transmission, reflection and absorption of each
# layer and the stack.
plotting.t_r_a_plots(stacks_list, xvalues = wavelengths)


######################## Wrapping up ########################
# Calculate and record the (real) time taken for simulation
elapsed = (time.time() - start)
hms     = str(datetime.timedelta(seconds=elapsed))
hms_string = 'Total time for simulation was \n \
    %(hms)s (%(elapsed)12.3f seconds)'% {
            'hms'       : hms,
            'elapsed'   : elapsed, }

python_log = open("python_log.log", "w")
python_log.write(hms_string)
python_log.close()

print hms_string
print '*****************************************'
print ''
```

3.12 Shear Transformations

```
"""
Here we introduce a shear transformation to shift layers relative to one
another in the plane.
"""

import time
import datetime
import numpy as np
import sys
from multiprocessing import Pool
sys.path.append("../backend/")

import objects
import materials
import plotting
from stack import *

start = time.time()
################ Simulation parameters ################

# Number of CPUs to use in simulation
num_cores = 5

# Remove results of previous simulations
plotting.clear_previous()
```

```
################ Light parameters ####################
azi_angles = np.linspace(0,20,5)
wl = 1600
light_list  = [objects.Light(wl, max_order_PWs = 2, theta = p, phi = 0.0) \
    for p in azi_angles]

################ Grating parameters ####################
period = 760

superstrate = objects.ThinFilm(period, height_nm = 'semi_inf',
    material = materials.Air, loss = False)

substrate  = objects.ThinFilm(period, height_nm = 'semi_inf',
    material = materials.Air, loss = False)

grating_1 = objects.NanoStruct('1D_array', period, small_d=period/2,
    diameter1=int(round(0.25*period)), diameter2=int(round(0.25*period)),
    height_nm = 150, inclusion_a = materials.Material(3.61 + 0.0j),
    inclusion_b = materials.Material(3.61 + 0.0j),
    background = materials.Material(1.46 + 0.0j),
    loss = True, make_mesh_now = True, force_mesh = False, lc_bkg = 0.1, lc2= 3.0)

grating_2 = objects.NanoStruct('1D_array', period, int(round(0.75*period)),
    height_nm = 2900, background = materials.Material(1.46 + 0.0j),
    inclusion_a = materials.Material(3.61 + 0.0j),
    loss = True, make_mesh_now = True, force_mesh = False, lc_bkg = 0.1, lc2= 3.0)

num_BMs = 60


def simulate_stack(light):
    ################ Evaluate each layer individually ##############
    sim_superstrate = superstrate.calc_modes(light)
    sim_substrate   = substrate.calc_modes(light)
    sim_grating_1   = grating_1.calc_modes(light, num_BMs = num_BMs)
    sim_grating_2   = grating_2.calc_modes(light, num_BMs = num_BMs)

    ###################### Evaluate structure ######################
    """ Now define full structure. Here order is critical and
        stack list MUST be ordered from bottom to top!
    """

    # Shear is relative to top layer (ie incident light) and in units of d.
    stack = Stack((sim_substrate, sim_grating_1, sim_grating_2, sim_superstrate), \
        shears = ([(0.1,0.0),(-0.3,0.1),(0.2,0.5)]) )
    stack.calc_scat(pol = 'TE')

    return stack


# Run in parallel across wavelengths.
pool = Pool(num_cores)
stacks_list = pool.map(simulate_stack, light_list)
# Save full simo data to .npz file for safe keeping!
np.savez('Simo_results', stacks_list=stacks_list)

plotting.t_r_a_plots(stacks_list)
```

```
######################## Wrapping up ########################
# Calculate and record the (real) time taken for simulation
elapsed = (time.time() - start)
hms     = str(datetime.timedelta(seconds=elapsed))
hms_string = 'Total time for simulation was \n \
    %(hms)s (%(elapsed)12.3f seconds)'% {
            'hms'       : hms,
            'elapsed'   : elapsed, }

python_log = open("python_log.log", "w")
python_log.write(hms_string)
python_log.close()

print hms_string
print '****************************************'
print ''
```

3.13 Ultrathin Absorption Limit - Varying n

```
"""
Simulating an ultrathin film with a range of real and imaginary refractive
indices. Can we reach the theoretical limit of 0.5 absorption?
"""

import time
import datetime
import numpy as np
import sys
from multiprocessing import Pool
sys.path.append("../backend/")

import objects
import materials
import plotting
from stack import *

start = time.time()

# Remove results of previous simulations.
plotting.clear_previous()

################ Simulation parameters ################
# Select the number of CPUs to use in simulation.
num_cores = 8

################ Light parameters ####################
wl     = 700
light  = objects.Light(wl, max_order_PWs = 0, theta = 0.0, phi = 0.0)

# The period must be consistent throughout a simulation!
period = 660

# Define each layer of the structure.
superstrate = objects.ThinFilm(period, height_nm = 'semi_inf',
    material = materials.Air, world_1d=True)
substrate   = objects.ThinFilm(period, height_nm = 'semi_inf',
```

```
    material = materials.Air, loss=False, world_1d=True)

n_min = 1
n_max = 10
num_n_re = 51
num_n_im = num_n_re
Re_n = np.linspace(n_min,n_max,num_n_re)
Im_n = np.linspace(n_max,n_min,num_n_im)
# Having lists run this way will ease plotting, as matshow plots from top left

def simulate_stack(Re):
    ################ Evaluate each layer individually ##############
    sim_superstrate = superstrate.calc_modes(light)
    sim_substrate   = substrate.calc_modes(light)

    # Re_stack = []
    # for Re in Re_n:
    Im_stack = []
    for Im in Im_n:
        TF_1 = objects.ThinFilm(period, height_nm = 10,
            material = materials.Material(Re + Im*1j))
        sim_TF_1 = TF_1.calc_modes(light)

        stack = Stack((sim_substrate, sim_TF_1, sim_superstrate))
        stack.calc_scat(pol = 'TM')

        Im_stack.append(stack)
        # Re_stack.append(Im_stack)

    return Im_stack

# Run wavelengths in parallel across num_cores CPUs using multiprocessing package.
pool = Pool(num_cores)
stacks_list = pool.map(simulate_stack, Re_n)
# # Save full simo data to .npz file for safe keeping!
# np.savez('Simo_results', stacks_list=stacks_list)

######################## Post Processing ########################

abs_mat = np.zeros((num_n_im,num_n_re))
for i in range(num_n_re):
    for j in range(num_n_im):
        abs_mat[j,i] = stacks_list[i][j].a_list[-1]

# Now plot as a function of Real and Imaginary refractive index.
# Requires a bit of manipulation of axis...
import matplotlib
matplotlib.use('pdf')
import matplotlib.pyplot as plt
fig = plt.figure()
linesstrength = 3
font = 18
ax1 = fig.add_subplot(1,1,1)
mat = ax1.matshow(abs_mat,cmap=plt.cm.hot)
cbar1 = plt.colorbar(mat, extend='neither',alpha=1)
ax1.xaxis.set_ticks_position('bottom')
ax1.set_xticks(np.linspace(n_min,(num_n-1),n_max))
ax1.set_yticks(np.linspace(n_min,(num_n-1),n_max))
```

```
ax1.set_xticklabels([str(i) for i in np.linspace(n_min,n_max,n_max-n_min+1)])
ax1.set_yticklabels([str(i) for i in np.linspace(n_max,n_min,n_max-n_min+1)])
ax1.set_xlabel('Re(n)',fontsize=font)
ax1.set_ylabel('Im(n)',fontsize=font)
plt.title('Absorption of %(h)5.1f nm thick film @ wl = %(wl)5.1f'% \
    {'h' : stacks_list[0][0].heights_nm()[0],'wl' : wl})
plt.savefig('ultrathin_limit')



######################## Wrapping up ########################
print '\n*****************************************'
# Calculate and record the (real) time taken for simulation,
elapsed = (time.time() - start)
hms     = str(datetime.timedelta(seconds=elapsed))
hms_string = 'Total time for simulation was \n \
    %(hms)s (%(elapsed)12.3f seconds)'% {
            'hms'       : hms,
            'elapsed'   : elapsed, }
print hms_string
print '*****************************************'
print ''

# and store this info.
python_log = open("python_log.log", "w")
python_log.write(hms_string)
python_log.close()
```

3.14 Varying a Layer of a Stack

```
"""
Simulating solar cell efficiency of nanohole array as a function of
substrate refractive indices (keeping geometry fixed).
We also average over a range of thicknesses to remove sharp Fabry-Perot resonances.
"""

import time
import datetime
import numpy as np
import sys
from multiprocessing import Pool
sys.path.append("../backend/")

import objects
import materials
import plotting
from stack import *

start = time.time()
################ Simulation parameters ################

# Number of CPUs to use in simulation
num_cores = 4

# Remove results of previous simulations
plotting.clear_previous()
```

```
################ Light parameters ####################
wl_1     = 310
wl_2     = 1127
no_wl_1  = 3
# Set up light objects
wavelengths = np.linspace(wl_1, wl_2, no_wl_1)
light_list  = [objects.Light(wl, max_order_PWs = 2, theta = 0.0, phi = 0.0) \
    for wl in wavelengths]


# Period must be consistent throughout simulation!!!
period = 550

cover  = objects.ThinFilm(period = period, height_nm = 'semi_inf',
    material = materials.Air, loss = True)

sub_ns = np.linspace(1.0,4.0,100)

NW_diameter = 480
NWs = objects.NanoStruct('1D_array', period, NW_diameter, height_nm = 2330,
    inclusion_a = materials.Si_c, background = materials.Air, loss = True,
    make_mesh_now = True, force_mesh = True, lc_bkg = 0.17, lc2= 2.5)

def simulate_stack(light):
    ################ Evaluate each layer individually ##############
    sim_cover = cover.calc_modes(light)
    sim_NWs   = NWs.calc_modes(light)

    # Loop over substrates
    stack_list = []
    for s in sub_ns:
        sub = objects.ThinFilm(period = period, height_nm = 'semi_inf',
        material = materials.Material(s + 0.0j), loss = False)
        sim_sub = sub.calc_modes(light)

        # Loop over heights to wash out sharp FP resonances
        average_t = 0
        average_r = 0
        average_a = 0

        num_h = 21
        for h in np.linspace(2180,2480,num_h):
            stackSub = Stack((sim_sub, sim_NWs, sim_cover), heights_nm = ([h]))
            stackSub.calc_scat(pol = 'TE')
            average_t += stackSub.t_list[-1]/num_h
            average_r += stackSub.r_list[-1]/num_h
            average_a += stackSub.a_list[-1]/num_h
        stackSub.t_list[-1] = average_t
        stackSub.r_list[-1] = average_r
        stackSub.a_list[-1] = average_a
        stack_list.append(stackSub)

    return stack_list


# Run in parallel across wavelengths.
pool = Pool(num_cores)
stacks_list = pool.map(simulate_stack, light_list)
```

```
# Save full simo data to .npz file for safe keeping!
np.savez('Simo_results', stacks_list=stacks_list)



######################## Plotting ########################
eta = []
for s in range(len(sub_ns)):
    stack_label = s # Specify which stack you are dealing with.
    stack1_wl_list = []
    for i in range(len(wavelengths)):
        stack1_wl_list.append(stacks_list[i][stack_label])
    sub_n = sub_ns[s]
    Efficiency = plotting.t_r_a_plots(stack1_wl_list, ult_eta=True,
        stack_label=stack_label, add_name = str(s))
    eta.append(100.0*Efficiency[0])
    # Dispersion of structured layer is the same for all cases.
    if s == 0:
        plotting.omega_plot(stack1_wl_list, wavelengths, stack_label=stack_label)

# Now plot as a function of substrate refractive index.
import matplotlib
matplotlib.use('pdf')
import matplotlib.pyplot as plt
fig = plt.figure()
linesstrength = 3
font = 18
ax1 = fig.add_subplot(1,1,1)
ax1.plot(sub_ns,eta, 'k-o', linewidth=linesstrength)
ax1.set_xlabel('Substrate refractive index',fontsize=font)
ax1.set_ylabel(r'$\eta$ (%)',fontsize=font)
plt.savefig('eta_substrates')

# Animate spectra as a function of substrates.
from os import system as ossys
delay = 30 # delay between images in gif in hundredths of a second
names = 'Total_Spectra_stack'
gif_cmd = 'convert -delay %(d)i +dither -layers Optimize -colors 16 \
%(n)s*.pdf %(n)s.gif'% {
'd' : delay, 'n' : names}
ossys(gif_cmd)
opt_cmd = 'gifsicle -O2 %(n)s.gif -o %(n)s-opt.gif'% {'n' : names}
ossys(opt_cmd)
rm_cmd = 'rm %(n)s.gif'% {'n' : names}
ossys(rm_cmd)



# Calculate and record the (real) time taken for simulation
elapsed = (time.time() - start)
hms     = str(datetime.timedelta(seconds=elapsed))
hms_string = 'Total time for simulation was \n \
    %(hms)s (%(elapsed)12.3f seconds)'% {
            'hms'       : hms,
            'elapsed'   : elapsed, }

python_log = open("python_log.log", "w")
python_log.write(hms_string)
```

```
python_log.close()

print '*****************************************'
print hms_string
print '*****************************************'
```

3.15 Convergence Testing

```
"""
    Replicate Fig 2a from Handmer Opt Lett 2010
"""


import time
import datetime
import numpy as np
import sys
from multiprocessing import Pool
sys.path.append("../backend/")

import objects
import materials
import plotting
from stack import *

start = time.time()
################ Simulation parameters ################

# Number of CPUs to use in simulation
num_cores = 8

# Remove results of previous simulations
plotting.clear_previous()

################ Light parameters ####################
wavelengths = np.linspace(1600,900,1)

BMs = [11,27,59,99,163,227,299,395,507,635,755,883,1059,1227,1419]
B = 0

for PWs in np.linspace(1,10,10):
    light_list  = [objects.Light(wl, max_order_PWs = PWs, theta = 28.0, phi = 0.0) for wl in wavelen

    ################ Grating parameters ####################
    period = 760

    superstrate = objects.ThinFilm(period, height_nm = 'semi_inf',
        material = materials.Air, loss = False)

    substrate  = objects.ThinFilm(period, height_nm = 'semi_inf',
        material = materials.Air, loss = False)

    grating_1 = objects.NanoStruct('1D_array', period, small_d=period/2,
        diameter1=int(round(0.25*period)), diameter2=int(round(0.25*period)), height_nm = 150,
        inclusion_a = materials.Material(3.61 + 0.0j), inclusion_b = materials.Material(3.61 + 0.0j),
        background = materials.Material(1.46 + 0.0j),
```

```
        loss = True, make_mesh_now = True, force_mesh = True, lc_bkg = 0.1, lc2= 3.0)

    grating_2 = objects.NanoStruct('1D_array', period, int(round(0.75*period)), height_nm = 2900,
        background = materials.Material(1.46 + 0.0j), inclusion_a = materials.Material(3.61 + 0.0j),
        loss = True, make_mesh_now = True, force_mesh = True, lc_bkg = 0.1, lc2= 3.0)

    num_BMs = BMs[B]+30
    B += 1


    def simulate_stack(light):

        ################ Evaluate each layer individually ##############
        sim_superstrate = superstrate.calc_modes(light)
        sim_substrate   = substrate.calc_modes(light)
        sim_grating_1   = grating_1.calc_modes(light, num_BMs = num_BMs)
        sim_grating_2   = grating_2.calc_modes(light, num_BMs = num_BMs)

        ###################### Evaluate structure ######################
        """ Now define full structure. Here order is critical and
            stack list MUST be ordered from bottom to top!
        """

        stack = Stack((sim_substrate, sim_grating_1, sim_grating_2, sim_superstrate))
        # stack = Stack((sim_substrate, sim_grating_2, sim_superstrate))
        stack.calc_scat(pol = 'TE')
        return stack



    # Run in parallel across wavelengths.
    pool = Pool(num_cores)
    stacks_list = pool.map(simulate_stack, light_list)
    # Save full simo data to .npz file for safe keeping!
    np.savez('Simo_results', stacks_list=stacks_list)


    additional_name = str(int(PWs))
    plotting.t_r_a_plots(stacks_list, add_name = additional_name)



####################### Wrapping up ########################
# Calculate and record the (real) time taken for simulation
elapsed = (time.time() - start)
hms     = str(datetime.timedelta(seconds=elapsed))
hms_string = 'Total time for simulation was \n \
    %(hms)s (%(elapsed)12.3f seconds)'% {
            'hms'       : hms,
            'elapsed'   : elapsed, }

# python_log = open("python_log.log", "w")
# python_log.write(hms_string)
# python_log.close()

print hms_string
print '*****************************************'
print ''
```

3.16 Extraordinary Optical Transmission

```
"""
Simulating Extraordinary Optical Transmission
as in H. Liu, P. Lalanne, Nature 452 2008 doi:10.1038/nature06762
"""

import time
import datetime
import numpy as np
import sys
from multiprocessing import Pool
sys.path.append("../backend/")

import objects
import materials
import plotting
from stack import *

start = time.time()
################ Simulation parameters ################

# Number of CPUs to use in simulation
num_cores = 16

# Remove results of previous simulations
plotting.clear_previous()

################ Light parameters ####################
wl_1     = 0.85*940
wl_2     = 1.15*940
no_wl_1  = 600
# Set up light objects
wavelengths = np.linspace(wl_1, wl_2, no_wl_1)
# wavelengths = np.array([785,788,790,792,795])
light_list  = [objects.Light(wl, max_order_PWs = 5, theta = 0.0, phi = 0.0) for wl in wavelengths]


#period must be consistent throughout simulation!!!
period = 940
diam1 = 266
NHs = objects.NanoStruct('2D_array', period, diam1, height_nm = 200,
    inclusion_a = materials.Air, background = materials.Au, loss = True,
    inc_shape = 'square',
    make_mesh_now = True, force_mesh = True, lc_bkg = 0.12, lc2= 5.0, lc3= 3.0)#lc_bkg = 0.08, lc2= 

strate  = objects.ThinFilm(period = period, height_nm = 'semi_inf',
    material = materials.Air, loss = False)

NH_heights = [200]
# num_h = 21
# NH_heights = np.linspace(50,3000,num_h)

def simulate_stack(light):
    ################ Evaluate each layer individually ##############
    sim_NHs    = NHs.calc_modes(light)
    sim_strate = strate.calc_modes(light)
```

```
# Loop over heights
    height_list = []
    for h in NH_heights:
        stackSub = Stack((sim_strate, sim_NHs, sim_strate), heights_nm = ([h]))
        stackSub.calc_scat(pol = 'TE')
        height_list.append(stackSub)

    return [height_list]


# Run in parallel across wavelengths.
pool = Pool(num_cores)
stacks_list = pool.map(simulate_stack, light_list)
# Save full simo data to .npz file for safe keeping!
np.savez('Simo_results', stacks_list=stacks_list)



######################## Plotting ########################
last_light_object = light_list.pop()

wls_normed = wavelengths/period

for h in range(len(NH_heights)):
    height = NH_heights[h]
    wl_list = []
    stack_label = 0
    for wl in range(len(wavelengths)):
        wl_list.append(stacks_list[wl][stack_label][h])
    mess_name = '_h%(h)i'% {'h'    : h, }
    plotting.EOT_plot(wl_list, wls_normed, add_name = mess_name, savetxt = True)
# Dispersion
plotting.omega_plot(wl_list, wavelengths)


# Calculate and record the (real) time taken for simulation
elapsed = (time.time() - start)
hms     = str(datetime.timedelta(seconds=elapsed))
hms_string = 'Total time for simulation was \n \
    %(hms)s (%(elapsed)12.3f seconds)'% {
            'hms'       : hms,
            'elapsed'   : elapsed, }

python_log = open("python_log.log", "w")
python_log.write(hms_string)
python_log.close()

print '****************************************'
print hms_string
print '****************************************'
print ''
```

3.17 Screen Sessions

`screen`

is an extremely useful little linux command. In the context of long-ish calculations it has two important applications; ensuring your calculation is unaffected if your connection to a remote machine breaks, and terminating calculations that have hung without closing the terminal. For more information see the manual:

```
$ man screen
```

or see online discussions here, and here.

The screen session or also called screen instance looks just like your regular terminal/putty, but you can disconnect from it (close putty, turn off your computer etc.) and later reconnect to the screen session and everything inside of this will have kept running. You can also reconnect to the session from a different computer via ssh.

3.17.1 Basic Usage

To install screen:

```
$ sudo apt-get install screen
```

To open a new screen session:

```
$ screen
```

We can start a new calculation here:

```
$ cd EMUstack/examples/
$ python simo_040-2D_array.py
```

We can then detach from the session (leaving everything in the screen running) by typing:

```
Ctrl +a
Ctrl +d
```

We can now monitor the processes in that session:

```
$ top
```

Where we note the numerous running python processes that EMUstack has started. Watching the number of processes is useful for checking if a long simulation is near completion (which is indicated by the number of processes dropping to less than the specified num_cores).

We could now start another screen and run some more calculations in this terminal (or do anything else). If we want to access the first session we 'reattach' by typing:

```
Ctrl +a +r
```

Or entering the following into the terminal:

```
$ screen -r
```

If there are multiple sessions use:

```
$ screen -ls
```

to get a listing of the sessions and their ID numbers. To reattach to a particular screen, with ID 1221:

```
$ screen -r 1221
```

To terminate a screen from within type:

```
Ctrl+d
```

Or, taking the session ID from the previous example:

```
screen -X -S 1221 kill
```

3.17.2 Terminating EMU stacks

If (for some estranged reason) a simulation hangs, we can kill all python instances upon the machine:

```
$ pkill python
```

If a calculation hangs from within a screen session one must first detach from that session then kill python. A more targeted way to kill processes is using their PID:

```
$ kill PID
```

Or if this does not suffice be a little more forceful:

```
$ kill -9 PID
```

The PID is found from one of two ways:

```
$ top
$ ps -fe | grep username
```

CHAPTER 4

Python Backend

4.1 objects module

objects.py is a subroutine of EMUstack that contains the NanoStruct, ThinFilm and Light objects. These represent the properties of a structured layer, a homogeneous layer and the incident light respectively.

class `objects.`**`EMUstack`**
Bases: `object`

class `objects.`**`Light`**(*wl_nm*, *max_order_PWs=2*, *k_parallel=None*, *theta=None*, *phi=None*, *n_inc=1.0*)
Bases: `object`

Represents the light incident on structure.

Incident angles may either be specified by *k_parallel* or by incident angles *theta* and *phi*, together with the refractive index *n_inc* of the incident medium.

wl_nm and *k_pll* are both in unnormalised units.

At normal incidence and TE polarisation the E-field is aligned with the y-axis.

At normal incidence some plane waves and Bloch modes become degenerate. This causes problems for the FEM solver and the ordering of the plane waves. To avoid this a small (1e-5) theta and phi are introduced.

Parameters **wl_nm** (*float*) – Wavelength, in nanometers.

Keyword Arguments

- **max_order_PWs** (*int*) – Maximum plane wave order to include.
- **k_parallel** (*tuple*) – The wave vector components (k_x, k_y) parallel to the interface planes. Units of nm^-1.
- **theta** (*float*) – Polar angle of incidence in degrees.
- **phi** (*float*) – Azimuthal angle of incidence in degrees measured from x-axis.

class objects.**NanoStruct**(*periodicity*, *period*, *diameter1*, *period_y=None*, *inc_shape='circle'*, *ellipticity=0.0*, *ff=0*, *ff_rand=False*, *small_space=None*, *edge_spacing=False*, *len_vertical=0*, *len_horizontal=0*, *background=<materials.Material object at 0x7f2b27203b10>*, *inclusion_a=<materials.Material object at 0x7f2b27203fd0>*, *inclusion_b=<materials.Material object at 0x7f2b27203990>*, *inclusion_c=<materials.Material object at 0x7f2b27203a10>*, *inclusion_d=<materials.Material object at 0x7f2b27203a50>*, *inclusion_e=<materials.Material object at 0x7f2b27211d90>*, *loss=True*, *height_nm=100.0*, *diameter2=0*, *diameter3=0*, *diameter4=0*, *diameter5=0*, *diameter6=0*, *diameter7=0*, *diameter8=0*, *diameter9=0*, *diameter10=0*, *diameter11=0*, *diameter12=0*, *diameter13=0*, *diameter14=0*, *diameter15=0*, *diameter16=0*, *gap=0*, *smooth=0*, *hyperbolic=False*, *world_1d=None*, *posx=0*, *posy=0*, *make_mesh_now=True*, *force_mesh=True*, *mesh_file='NEED_FILE.mail'*, *lc_bkg=0.09*, *lc2=1.0*, *lc3=1.0*, *lc4=1.0*, *lc5=1.0*, *lc6=1.0*, *plotting_fields=False*, *plot_real=1*, *plot_imag=0*, *plot_abs=0*, *plot_field_conc=False*, *plt_msh=True*)

Bases: object

Represents a structured layer.

Parameters

- **periodicity** (*str*) – Either 1D or 2D structure '1D_array', '2D_array'.
- **period** (*float*) – The period of the unit cell in nanometers.
- **diameter1** (*float*) – The diameter of the inclusion in nm.

Keyword Arguments

- **period_y** (*float*) – The period of the unit cell in the y-direction. If None, period_y = period.
- **inc_shape** (*str*) – Shape of inclusions that have template mesh, currently; 'circle', 'ellipse', 'square', 'ring', 'SRR', 'dimer', 'square_dimer', 'strip_circle', 'strip_square'.
- **ellipticity** (*float*) – If != 0, inclusion has given ellipticity, with b = diameter, a = diameter-ellipticity * diameter. NOTE: only implemented for a single inclusion.
- **len_vertical** (*float*) – Vertical length of split ring resonator (if inc_shape = 'SRR').
- **len_horizontal** (*float*) – Horizontal length of split ring resonator (if inc_shape = 'SRR').
- **diameter2-16 (float): The diameters of further inclusions in nm. Implemented up to diameter6 for 1D_arrays.**
- **gap** (*float*) – The dimer gap in nm. (if inc_shape = 'dimer' or 'square_dimer').
- **smooth** (*float*) – smoothness of square_dimer angles, between 0 (sharp). and 1 (circle). (if inc_shape = square_dimer').
- **inclusion_a** – A :Material: instance for first inclusion, specified as dispersive refractive index (eg. materials.Si_c) or nondispersive complex number (eg. Material(1.0 + 0.0j)).
- **inclusion_b** – A :Material: instance for the second inclusion medium.
- **inclusion_c** – A :Material: instance for the third inclusion medium.
- **inclusion_d** – A :Material: instance for the fourth inclusion medium.
- **inclusion_e** – A :Material: instance for the fifth inclusion medium.
- **background** – A :Material: instance for the background medium.
- **loss** (*bool*) – If False, Im(n) = 0, if True n as in :Material: instance.

- **height_nm** (*float*) – The thickness of the layer in nm or 'semi_inf' for a semi-infinite layer.
- **hyperbolic** (*bool*) – If True FEM looks for Eigenvalues around n**2 * k_0**2 rather than the regular n**2 * k_0**2 - alpha**2 - beta**2.
- **world_1d** (*bool*) – Does the rest of the stack have exclusively 1D periodic structures and homogeneous layers? If True we use the set of 1D diffraction order PWs. Defaults to True for '1D_array', and False for '2D_array'.
- **ff** (*float*) – The fill fraction of the inclusions. If non-zero, the specified diameters are overwritten s.t. given ff is achieved, otherwise ff is calculated from parameters and stored in self.ff.
- **ff_rand** (*bool*) – If True, diameters overwritten with random diameters, s.t. the ff is as assigned. Must provide non-zero dummy diameters.
- **posx** (*float*) – Shift NWs laterally towards center (each other), posx is a fraction of the distance possible before NWs touch.
- **posy** (*float*) – Shift NWs vertically towards center (each other), posx is a fraction of the distance possible before NWs touch.
- **small_space** (*float*) – Only for 1D_arrays with 2 interleaved inclusions. Sets distance between edges of inclusions. By default (d_in_nm - diameter1 - diameter2) / 2. The smaller distance is on the, which left of center (inclusion_a remains centered).
- **edge_spacing** (*bool*) – For 1D_array with >= 3 inclusions. Space inclusion surfaces by equal separations. Else their centers will be equally spaced.
- **make_mesh_now** (*bool*) – If True, program creates a FEM mesh with provided :NanoStruct: parameters. If False, must provide mesh_file name of existing .mail that will be run despite :NanoStruct: parameters.
- **force_mesh** (*bool*) – If True, a new mesh is created despite existence of mesh with same parameter. This is used to make mesh with equal period etc. but different lc refinement.
- **mesh_file** (*str*) – If using a set premade mesh give its name including .mail if 2D_array (eg. 600_60.mail), or .txt if 1D_array. It must be located in backend/fortran/msh/
- **lc_bkg** (*float*) – Length constant of meshing of background medium (smaller = finer mesh)
- **lc2** (*float*) – factor by which lc_bkg will be reduced on inclusion surfaces; lc_surface = cl_bkg / lc2.
- **lc3-6' (float): factor by which lc_bkg will be reduced at center of inclusions.**
- **plotting_fields** (*bool*) – Unless set to true field data deleted. Also plots modes (ie. FEM solutions) in gmsh format. Plots epsilon*|E|^2 & choice of real/imag/abs of x,y,z components & field vectors. Fields are saved as gmsh files, but can be converted by running the .geo file found in Bloch_fields/PNG/
- **plot_real** (*bool*) – Choose to plot real part of modal fields.
- **plot_imag** (*bool*) – Choose to plot imaginary part of modal fields.
- **plot_abs** (*bool*) – Choose to plot absolute value of modal fields.
- **plt_msh** (*bool*) – Save a plot of the 1D array geometry.

calc_modes(*light*, ***args*)

Run a simulation to find the NanoStruct's modes.

Parameters

- **light** (*Light instance*) – Represents incident light.

- **args** (*dict*) – Options to pass to :Simmo.calc_modes:.

Returns

Simmo object

make_mesh()

class objects.**ThinFilm**(*period*, *period_y=None*, *height_nm=1.0*, *num_pw_per_pol=0*, *world_1d=False*, *material=<materials.Material object at 0x7f2b27211d50>*, *loss=True*)

Bases: object

Represents an unstructured homogeneous film.

Parameters **period** (*float*) – Artificial period imposed on homogeneous film to give consistently defined plane waves in terms of diffraction orders of structured layers.

Keyword Arguments

- **period_y** (*float*) – The period of the unit cell in the y-direction. If None, period_y = period.
- **height_nm** (*float*) – The thickness of the layer in nm or 'semi_inf' for a semi-infinte layer.
- **num_pw_per_pol** (*int*) – The number of plane waves per polarisation.
- **world_1d** (*bool*) – Does the rest of the stack have exclusively 1D periodic structures and homogeneous layers? If True we use the set of 1D diffraction order PWs.
- **material** – A :Material: instance specifying the n of the layer and related methods.
- **loss** (*bool*) – If False sets Im(n) = 0, if True leaves n as is.

calc_modes(*light*)

Run a simulation to find the ThinFilm's modes.

Parameters

- **light** (*Light instance*) – Represents incident light.
- **args** (*dict*) – Options to pass to :Anallo.calc_modes:.

Returns

Anallo object

objects.**calculate_ff**(*inc_shape*, *d*, *dy*, *a1*, *a2=0*, *a3=0*, *a4=0*, *a5=0*, *a6=0*, *a7=0*, *a8=0*, *a9=0*, *a10=0*, *a11=0*, *a12=0*, *a13=0*, *a14=0*, *a15=0*, *a16=0*, *el1=0*)

Calculate the fill fraction of the inclusions.

Parameters

- **inc_shape** (*str*) – shape of the inclusions.
- **d** (*float*) – period of structure, in same units as a1-16.
- **dy** (*float*) – period of structure along y-axis, in same units as a1-16.
- **a1** (*float*) – diameter of inclusion 1, in same units as d.

Keyword Arguments

- **a2-16 (float): diameters of further inclusions.**
- **el1** (*float*) – ellipticity of inclusion 1.

objects.**dec_float_str**(*dec_float*)

Convert float with decimal point into string with '_' in place of '.'

4.2 materials module

materials.py is a subroutine of EMUstack that defines Material objects, these represent dispersive lossy refractive indices and possess methods to interpolate n from tabulated data.

class `materials.`**`Material`**`(`*`n`*`)`

Bases: `object`

Represents a material with a refractive index n.

If the material is dispersive, the refractive index at a given wavelength is calculated by linear interpolation from the initially given data *n*. Materials may also have *n* calculated from a Drude model with input parameters.

Parameters **n** – Either a scalar refractive index, an array of values *(wavelength, n)*, or *(wavelength, real(n), imag(n))*, or omega_p, omega_g, eps_inf for Drude model.

Currently included materials are;

Semiconductors	**Metals**	**Transparent oxides**
Si_c	Au	TiO2
Si_a	Au_Palik	TiO2_anatase
SiO2	Ag	ITO
CuO	Ag_Palik	ZnO
CdTe	Cu	SnO2
FeS2	Cu_Palik	FTO_Wenger
Zn3P2		FTO_Wengerk5
AlGaAs		
Al2O3		
Al2O3_PV		
GaAs		
InGaAs	**Drude**	**Other**
Si3N4	Au_drude	Air
MgF2		H2O
InP		Glass
InAs		Spiro
GaP		Spiro_nk
Ge		
AlN		
GaN		
MoO3		
ZnS		
AlN_PV		
		Experimental incl.
		CH3NH3PbI3
		Sb2S3

Continued on next page

Table 4.1 – continued from previous page

		Sb2S3_ANU2014
		Sb2S3_ANU2015
		GO_2014
		GO_2015
		rGO_2015
		SiON_Low
		SiON_High
		Low_Fe_Glass
		Perovskite_00
		Perovskite
		Ge_Doped

__getstate__()
Can't pickle self._n, so remove it from what is pickled.

__setstate__(*d*)
Recreate self._n when unpickling.

n(*wl_nm*)
Return n for the specified wavelength.

materials.**plot_n_data**(*data_name*)

4.3 mode_calcs module

mode_calcs.py is a subroutine of EMUstack that contains methods to calculate the modes of a given layer, either analytically (class 'Anallo') or from the FEM routine (class 'Simmo').

class mode_calcs.**Anallo**(*thin_film*, *light*)
Bases: mode_calcs.Modes

Interaction of one :Light: object with one :ThinFilm: object.

Like a :Simmo:, but for a thin film, and calculated analytically.

Z()
Return the wave impedance as a 1D array.

calc_kz()
Return a sorted 1D array of grating orders' kz.

calc_modes()
Calculate the modes of homogeneous layer analytically.

k()
Return the normalised wavenumber in the background material.

n()
Return refractive index of an object at its wavelength.

specular_incidence(*pol='TE'*)
Return a vector of plane wave amplitudes corresponding to specular incidence in the specified polarisation.

i.e. all elements are 0 except the zeroth order.

class mode_calcs.**EMUstack**
Bases: object

class mode_calcs.**Modes**
Bases: object

Super-class from which Simmo and Anallo inherit common functionality.

air_ref()
Return an :Anallo: for air for the same :Light: as this.

calc_1d_grating_orders(*max_order*)
Return the grating order indices px and py, unsorted.

calc_2d_grating_orders(*max_order*)
Return the grating order indices px and py, unsorted.

k_pll_norm()

prop_fwd(*height_norm*)
Return the matrix P corresponding to forward propagation/decay.

shear_transform(*coords*)
Return the matrix Q corresponding to a shear transformation to coordinats coords.

wl_norm()
Return normalised wavelength (wl/period).

class mode_calcs.**Simmo**(*structure*, *light*)
Bases: mode_calcs.Modes

Interaction of one :Light: object with one :NanoStruc: object.

Inherits knowledge of :NanoStruc:, :Light: objects Stores the calculated modes of :NanoStruc: for illumination by :Light:

calc_modes(*num_BMs=None*)
Run a Fortran FEM calculation to find the modes of a structured layer.

mode_calcs.**r_t_mat**(*lay1*, *lay2*)
Return R12, T12, R21, T21 at an interface between lay1 and lay2.

mode_calcs.**r_t_mat_anallo**(*an1*, *an2*)
Returns R12, T12, R21, T21 at an interface between thin films.

R12 is the reflection matrix from Anallo 1 off Anallo 2

The sign of elements in T12 and T21 is fixed to be positive, in the eyes of *numpy.sign*

mode_calcs.**r_t_mat_tf_ns**(*an1*, *sim2*)
Returns R12, T12, R21, T21 at an1-sim2 interface.

Based on: Dossou et al., JOSA A, Vol. 29, Issue 5, pp. 817-831 (2012)

But we use Zw = 1/(Zcr X) instead of X, so that an1 does not have to be free space.

4.4 stack module

stack.py is a subroutine of EMUstack that contains the Stack object, which takes layers with known scattering matrices and calculates the net scattering matrices of the multilayered stack.

class `stack.`**`Stack`**(*layers*, *heights_nm=None*, *shears=None*)

Bases: `object`

Represents a stack of layers evaluated at one frequency.

This includes the semi-infinite input and output layers.

Parameters

- **layers** (*tuple*) – :ThinFilm:s and :NanoStruct:s ordered from top to bottom layer.
- **heights_nm** (*tuple*) – the heights of the inside layers, i.e., all layers except for the top and bottom. This overrides any heights specified in the :ThinFilm: or :NanoStruct: objects.
- **shears** (*tuple*) – the in-plane coordinates of each layer, including semi-inf layers in unit cell units (i.e. 0-1). e.g. ([0.0, 0.3], [0.1, 0.1], [0.2, 0.5]) for '2D_array' e.g. ([0.0], [0.1], [0.5]) for '1D_array'. Only required if wish to shift layers relative to each other. Only relative difference matters.

Note that: scattering matrices, are organised as | TE -> TE | TM -> TE | | TE -> TM | TM -> TM |

`calc_scat`(*pol='TE'*, *incoming_amplitudes=None*, *calc_fluxes=True*, *save_scat_list=False*)

Calculate the transmission and reflection matrices of the stack.

In relation to the FEM mesh the polarisation is orientated, - along the y axis for TE - along the x axis for TM at normal incidence (polar angle theta = 0, azimuthal angle phi = 0).

Keyword Arguments

- **pol** (*str*) – Polarisation for which to calculate transmission & reflection.
- **incoming_amplitudes** (*int*) – Which incoming PW order to give 1 unit of energy. If None the 0th order PW is selected.
- **calc_fluxes** (*bool*) – Calculate energy fluxes. Only possible if top layer is a ThinFilm.
- **save_scat_list** (*bool*) – If True, save tnet_list, rnet_list as property of stack for later access.

`heights_nm`()

Update heights of each layer to those given in Keyword Arg 'heights_nm'. If no heights specified in Stack, the heights of each layer object are used.

`heights_norm`()

Normalise heights by the array period.

`structures`()

Return :NanoStruct: or :ThinFilm: object of each layer.

total_height()
Calculate total thickness of stack.

4.5 plotting module

plotting.py is a subroutine of EMUstack that contains numerous plotting routines.

plotting.**BM_amplitudes**(*stacks_list*, *xvalues=None*, *chosen_BMs=None*, *lay_interest=1*, *up_and_down=True*, *add_height=None*, *add_name=''*, *save_pdf=True*, *save_npz=False*, *save_txt=False*)
Plot the amplitudes of Bloch modes in selected layer.

Parameters **stacks_list** (*list*) – Stack objects containing data to plot.

Keyword Arguments

- **xvalues** (*list*) – The values stacks_list is to be plotted as a function of.
- **chosen_BMs** (*list*) – Bloch Modes to include, identified by their indices in the scattering matrices (order most propagating to most evanescent) eg. [0,2,4].
- **lay_interest** (*int*) – The index in stacks_list of the layer in which amplitudes are calculated.
- **up_and_down** (*bool*) – Average the amplitudes of up & downward propagating modes. Else include only downward in all layers except for the superstrate where include only upward.
- **add_height** (*float*) – Print the heights of :Stack: layer in title.
- **add_name** (*str*) – Add add_name to title.
- **save_pdf** (*bool*) – If True save spectra as pdf files. True by default.
- **save_npz** (*bool*) – If True, saves lists of BM amplitudes to file.
- **save_txt** (*bool*) – If True, saves lists of BM amps to txt file.

plotting.**Bloch_fields_1d**(*stacks_list*, *lay_interest=None*)
Plot Bloch mode fields along the x axis.

Args: stacks_list (list): Stack objects containing data to plot.

Keyword Args: lay_interest (int): the index of the layer considered within the stack. Must be a 1D_array NanoStruct layer. By default routine finds all such layers.

plotting.**EOT_plot**(*stacks_list*, *wavelengths*, *pol='TM'*, *params_layer=1*, *add_name=''*, *savetxt=False*)
Plot T_{00} as in Martin-Moreno PRL 86 2001.

Parameters **pol** (*str*) – Take the (0,0) component for TE->TE scattering or the (0,0) component for TM->TM scattering.

plotting.**E_conc_plot**(*stacks_list*, *which_layer*, *which_modes*, *wavelengths*, *params_layer=1*, *stack_label=1*)

Plots the energy concentration (epsilon E_cyl / epsilon E_cell) of given layer.

Parameters

- **stacks_list** (*list*) – Stack objects containing data to plot.
- **which_layer** (*int*) – The index in stacks_list of the layer for which the energy concentration is to be calculated.
- **which_modes** (*list*) – Indices of Bloch modes for which to calculate the energy concentration.
- **wavelengths** (*list*) – The wavelengths corresponding to stacks_list.

Keyword Arguments

- **params_layer** (*int*) – The index in stacks_list of the layer for which the geometric parameters are put in the title of the plots.
- **stack_label** (*int*) – Label to differentiate plots of different :Stack:s.

plotting.**Fabry_Perot_res**(*stacks_list*, *freq_list*, *kx_list*, *f_0*, *k_0*, *lay_interest=1*)

Calculate the Fabry-Perot resonance condition for a resonances within a layer.

This is equivalent to finding the slab waveguide modes of the layer.

Parameters

- **stacks_list** (*list*) – Stack objects containing data to plot.
- **freq_list** (*list*) – Frequencies included.
- **kx_list** (*list*) – In-plane wavenumbers included.
- **f_0** (*list*) – Frequency w.r.t. which axis is normalised.
- **k_0** (*list*) – In-plane wavenumber w.r.t. which axis is normalised.

Keyword Arguments **lay_interest** (*int*) – The index in stacks_list of the layer of which F-P resonances are calculated.

plotting.**J_sc_eta_NO_plots**(*stacks_list*, *wavelengths*, *params_layer=1*, *active_layer_nu=1*, *stack_label=1*, *add_name=''*)

Calculate J_sc & ultimate efficiency but do not save or plot spectra.

Parameters

- **stacks_list** (*list*) – Stack objects containing data to plot.
- **wavelengths** (*list*) – The wavelengths corresponding to stacks_list.

Keyword Arguments

- **params_layer** (*int*) – The index in stacks_list of the layer for which the geometric parameters are put in the title of the plots.
- **active_layer_nu** (*int*) – The index in stacks_list (from bottom) of the layer for which the ult_eta and/or J_sc are calculated.
- **stack_label** (*int*) – Label to differentiate plots of different :Stack:s.
- **add_name** (*str*) – Add add_name to title.

plotting.**J_short_circuit**(*active_abs*, *wavelengths*, *params_2_print=''*, *stack_label=''*, *add_name=''*)

Calculate the short circuit current J_sc under ASTM 1.5 illumination. Assuming every absorbed photon produces a pair of charge carriers.

plotting.**PW_amplitudes**(*stacks_list*, *xvalues=None*, *chosen_PWs=None*, *lay_interest=0*, *up_and_down=True*, *add_height=None*, *add_name=''*, *save_pdf=True*, *save_npz=False*, *save_txt=False*)

Plot the amplitudes of plane wave orders in selected layer.

Assumes dealing with 1D grating and only have 1D diffraction orders. Takes the average of up & downward propagating modes.

Parameters **stacks_list** (*list*) – Stack objects containing data to plot.

Keyword Arguments

- **xvalues** (*list*) – The values stacks_list is to be plotted as a function of.
- **chosen_PWs** (*list*) – PW diffraction orders to include. eg. [-1,0,2]. If 'None' are given all are plotted.
- **lay_interest** (*int*) – The index in stacks_list of the layer in which amplitudes are calculated.
- **up_and_down** (*bool*) – Average the amplitudes of up & downward propagating modes. Else include only downward in all layers except for the superstrate where include only upward.
- **add_height** (*float*) – Print the heights of :Stack: layer in title.
- **add_name** (*str*) – Add add_name to title.
- **save_pdf** (*bool*) – If True save spectra as pdf files. True by default.
- **save_npz** (*bool*) – If True, saves lists of PW amplitudes to file.
- **save_txt** (*bool*) – If True, saves lists of PW amps to txt file.

plotting.**clear_previous**()

Delete all files of specified type as well as field directories.

plotting.**evanescent_merit**(*stacks_list*, *xvalues=None*, *chosen_PWs=None*, *lay_interest=0*, *add_height=None*, *add_name=''*, *save_pdf=True*, *save_txt=False*)

Plot a figure of merit for the 'evanescent-ness' of excited fields.

Assumes dealing with 1D grating and only have 1D diffraction orders.

Parameters **stacks_list** (*list*) – Stack objects containing data to plot.

Keyword Arguments

- **xvalues** (*list*) – The values stacks_list is to be plotted as a function of.
- **chosen_PWs** (*list*) – PW diffraction orders to include. eg. [-1,0,2].
- **lay_interest** (*int*) – The index in stacks_list of the layer in which amplitudes are calculated.
- **add_height** (*float*) – Print the heights of :Stack: layer in title.
- **add_name** (*str*) – Add add_name to title.
- **save_pdf** (*bool*) – If True save spectra as pdf files. True by default.
- **save_txt** (*bool*) – If True, saves average value of mean PW order to file.

plotting.**extinction_plot**(*t_spec*, *wavelengths*, *params_2_print*, *stack_label*, *add_name*)

Plot extinction ratio in transmission extinct = log_10(1/t).

plotting.**field_values**(*stacks_list, lay_interest=0, xyz_values=[(0.1, 0.1, 0.1)]*)

Save electric field values at given x-y-z points. Points must be within a ThinFilm layer. In txt file fields are given as Re(Ex) Im(Ex) Re(Ey) Im(Ey) Re(Ez) Im(Ez) |E|

Args: stacks_list (list): Stack objects containing data to plot.

Keyword Args: lay_interest (int): the index of the layer considered within the stack. Must be a ThinFilm layer.

xyz_values (list): list of distances in normalised units of (d) from top surface of layer at which to calculate fields. For semi-inf substrate then z_value is distance from top of this layer (i.e. bottom interface of stack).

plotting.**fields_3d**(*stacks_list*, *lay_interest=1*)

Plot fields in 3D using gmsh.

Args: stacks_list (list): Stack objects containing data to plot.

Keyword Args: lay_interest (int): the index of the layer considered within the stack.

plotting.**fields_in_plane**(*stacks_list, lay_interest=1, z_values=[0.1, 3.6], nu_calc_pts=51*)

Plot fields in the x-y plane at chosen values of z.

Args: stacks_list (list): Stack objects containing data to plot.

Keyword Args: lay_interest (int): the index of the layer considered within the stack.

z_values (float): distance in nm from bottom surface of layer at which to calculate fields. If layer is semi-inf substrate then z_value is distance from top of this layer (i.e. bottom interface of stack).

nu_calc_pts (int): fields are calculated over a mesh of nu_calc_pts * nu_calc_pts points.

plotting.**fields_interpolator_in_plane**(*pstack*, *lay_interest=1*, *z_value=0.1*)

Returns linear interpolators in the x-y plane at chosen value of z for Re[Ex],Re[Ey],Re[Ez],Im[Ex],Im[Ey],Im[Ez],|E|.

Requires matplotlib v1.3.0 or later.

Args: pstack: Stack object (not a list!!!) containing data to plot.

Keyword Args: lay_interest (int): the index of the layer considered within the stack.

z_value (float): distance in nm from bottom surface of layer at which to calculate fields. If layer is semi-inf substrate then z_value is distance from top of this layer (i.e. bottom interface of stack).

plotting.**fields_vertically**(*stacks_list*, *factor_pts_vert=31*, *nu_pts_hori=41*, *semi_inf_height=1.0*, *gradient=None*, *no_incoming=False*, *add_name=''*, *force_eq_ratio=False*, *colour_res=30*, *plot_boundaries=True*)

Plot fields in the x-y plane at chosen values of z, where z is calculated from the bottom of chosen layer.

Args: stacks_list (list): Stack objects containing data to plot.

Keyword Args: factor_pts_vert (int): sampling factor for fields vertically. Calculated as factor_pts_vert * (epsilon*h/wl).

nu_pts_hori (int): in-plane fields are calculated over a mesh of nu_pts_hori * nu_pts_hori points.

semi_inf_height (float): distance to which fields are plotting in semi-infinite (sub)superstrates.

gradient (float): further slices calculated with given gradient and -gradient. It is entitled 'specified_diagonal_slice'. These slices are only calculated for ThinFilm layers.

no_incoming (bool): if True, plots fields in superstrate in the absence of the incident driving field (i.e. only showing upward propagating scattered field).

add_name (str): concatenate add_name to title.

force_eq_ratio (bool): each layer plotted on equal space.

colour_res (int): number of colour intervals to use.

plot_boundaries (bool): plot boundaries of inclusions.

plotting.**gen_params_string**(*stack*, *layer=1*)
Generate the string of simulation info that is to be printed at the top of plots.

plotting.**layers_plot**(*spectra_name*, *spec_list*, *xvalues*, *xlabel*, *total_h*, *params_2_print*, *stack_label*, *add_name*, *save_pdf*, *save_txt*, *label_eV*, *set_y_lim*)
Plots one type of spectrum across all layers.

Is called from t_r_a_plots.

plotting.**layers_print**(*spectra_name*, *spec_list*, *wavelengths*, *total_h*, *stack_label=1*, *add_name=''*)
Save spectra to text files.

Is called from t_r_a_write_files.

plotting.**max_n**(*stacks_list*)
Find maximum refractive index n in stacks_list.

plotting.**omega_plot**(*stacks_list*, *wavelengths*, *params_layer=1*, *stack_label=1*)
Plots the dispersion diagram of each layer in one plot. k_z has units nm^-1.

Parameters

- **stacks_list** (*list*) – Stack objects containing data to plot.
- **wavelengths** (*list*) – The wavelengths corresponding to stacks_list.

Keyword Arguments

- **params_layer** (*int*) – The index in stacks_list of the layer for which the geometric parameters are put in the title of the plots.
- **stack_label** (*int*) – Label to differentiate plots of different :Stack:s.

plotting.**t_func_k_plot_1D**(*stacks_list*, *lay_interest=0*, *pol='TE'*)
PW amplitudes in transmission as a function of their in-plane k-vector.

Parameters **stacks_list** (*list*) – Stack objects containing data to plot.

Keyword Arguments

- **lay_interest** (*int*) – The index in stacks_list of the layer in which amplitudes are calculated.
- **pol** (*str*) – Include transmission in Which polarisation.

plotting.**t_r_a_plots**(*stacks_list*, *xvalues=None*, *xlabel=''*, *params_layer=1*, *active_layer_nu=1*, *stack_label=1*, *ult_eta=False*, *J_sc=False*, *weight_spec=False*, *extinct=False*, *add_height=0*, *add_name=''*, *save_pdf=True*, *save_txt=False*, *set_y_lim=True*, *label_eV=False*)
Plot t, r, a for each layer & in total.

Parameters **stacks_list** (*list*) – Stack objects containing data to plot.

Keyword Arguments

- **xvalues** (*list*) – The values stacks_list is to be plotted as a function of.

- **params_layer** (*int*) – The index in stacks_list of the layer for which the geometric parameters are put in the title of the plots.
- **active_layer_nu** (*int*) – The index in stacks_list (from bottom) of the layer for which the ult_eta and/or J_sc are calculated.
- **stack_label** (*int*) – Label to differentiate plots of different :Stack:s.
- **ult_eta** (*bool*) – If True; calculate the 'ultimate efficiency'.
- **J_sc** (*bool*) – If True; calculate the idealised short circuit current.
- **weight_spec** (*bool*) – If True; plot t, r, a spectra weighted by the ASTM 1.5 solar spectrum.
- **extinct** (*bool*) – If True; calculate the extinction ratio in transmission.
- **add_height** (*float*) – Print the heights of :Stack: layer in title.
- **add_name** (*str*) – Add add_name to title.
- **save_pdf** (*bool*) – If True; save spectra as pdf files. True by default.
- **save_txt** (*bool*) – If True; save spectra data to text files.
- **set_y_lim** (*bool*) – If True; set y limits to (0,1).
- **label_eV** (*bool*) – If True; add energy label in eV.

plotting.**t_r_a_plots_subs**(*stacks_list*, *wavelengths*, *period*, *sub_n*, *params_layer=1*, *active_layer_nu=1*, *stack_label=1*, *ult_eta=False*, *J_sc=False*, *weight_spec=False*, *extinct=False*, *add_name=''*)

Plot t, r, a indicating Wood anomalies in substrate for each layer & total.

Parameters

- **stacks_list** (*list*) – Stack objects containing data to plot.
- **wavelengths** (*list*) – The wavelengths corresponding to stacks_list.
- **period** (*float*) – Period of :Stack:s.
- **sub_n** (*float*) – Refractive index of the substrate in which Wood anomalies are considered.

Keyword Arguments

- **params_layer** (*int*) – The index in stacks_list of the layer for which the geometric parameters are put in the title of the plots.
- **active_layer_nu** (*int*) – The index in stacks_list (from bottom) of the layer for which the ult_eta and/or J_sc are calculated.
- **stack_label** (*int*) – Label to differentiate plots of different :Stack:s.
- **ult_eta** (*bool*) – If True, calculate the 'ultimate efficiency'.
- **J_sc** (*bool*) – If True, calculate the idealised short circuit current.
- **weight_spec** (*bool*) – If True, plot t, r, a spectra weighted by the ASTM 1.5 solar spectrum.
- **extinct** (*bool*) – If True, calculate the extinction ratio in transmission.

  add_name (str): Add add_name to title.

plotting.**t_r_a_write_files**(*stacks_list*, *wavelengths*, *stack_label=1*, *add_name=''*)

Save t, r, a for each layer & total in text files.

Parameters

- **stacks_list** (*list*) – Stack objects containing data to plot.

- **wavelengths** (*list*) – The wavelengths corresponding to stacks_list.

Keyword Arguments

- **stack_label** (*int*) – Label to differentiate plots of different :Stack:s.
- **add_name** (*str*) – Add add_name to title.

plotting.**tick_function**(*energies*)
Convert energy in eV into wavelengths in nm

plotting.**total_tra_plot**(*plot_name*, *a_spec*, *t_spec*, *r_spec*, *xvalues*, *xlabel*, *params_2_print*, *stack_label*, *add_name*, *label_eV*, *set_y_lim*)
Plots total t, r, a spectra on one plot.

Is called from t_r_a_plots, t_r_a_plots_subs

plotting.**total_tra_plot_subs**(*plot_name*, *a_spec*, *t_spec*, *r_spec*, *wavelengths*, *params_2_print*, *stack_label*, *add_name*, *period*, *sub_n*)
Plots total t, r, a spectra with lines at first 6 Wood anomalies.

Is called from t_r_a_plots_subs

plotting.**ult_efficiency**(*active_abs*, *wavelengths*, *bandgap_wl=None*, *params_2_print=''*, *stack_label=''*, *add_name=''*)
Calculate the photovoltaic ultimate efficiency achieved in the specified active layer.

For definition see Sturmberg et al., Optics Express, Vol. 19, Issue S5, pp. A1067-A1081 (2011).

Parameters **bandgap_wl** (*float*) – allows you to set the wavelength equivalent to the bandgap. Else it is assumed to be the maximum wavelength simulated.

plotting.**vis_matrix**(*scat_mat*, *add_name=''*, *max_scale=None*, *only_real=False*)
Plot given matrix as a greyscale image.

Parameters **scat_mat** (*np.matrix*) – A matrix.

Keyword Arguments

- **add_name** (*str*) – Add add_name to title.
- **max_scale** (*float*) – Limit maximum amplitude shown.
- **only_real** (*bool*) – Only plot the real part of matrix.

plotting.**vis_scat_mats**(*scat_mat*, *nu_prop_PWs=0*, *wl=None*, *add_name=''*, *max_scale=None*)
Plot given scattering matrix as greyscale images.

Parameters **scat_mat** (*np.matrix*) – A scattering matrix, which is organised as | TE -> TE | TM -> TE | | TE -> TM | TM -> TM |

Keyword Arguments

- **nu_prop_PWs** (*int*) – Number of propagating PWs.
- **wl** (*int*) – Index in case of calling in a loop.
- **add_name** (*str*) – Add add_name to title.
- **max_scale** (*float*) – Limit maximum amplitude shown.

plotting.**zeros_int_str**(*zero_int*)
Convert integer into string with '0' in place of ' '.

CHAPTER 5

Fortran Backends

The intention of EMUstack is that the Fortran FEM routines are essentially black boxes. They are called from mode_calcs.py and return the modes (Eigenvalues) of a structured layer, as well as some matrices of overlap integrals that are then used to compute the scattering matrices.

There are however a few important things to know about the workings of these routines.

5.1 1D FEM Mode Solver

5.1.1 1D Mesh

1D FEM mesh are created by the python subroutine objects.make_mesh() and passed directly into the fortran routine 'py_calc_modes.f'. The only parameter that influences this process is **'lc_bkg'**, where **1 / lc_bkg** is the number of FEM elements that the unit cell is divided into.

For a single inclusion the mesh is simply:

```
|                    period                    |

|--------------------| 1 |--------------------|
```

where the inclusion has 'diameter1' as is made of material 'inclusion_a'.

For a grating with 2 inclusions in the unit cell the spacing between the surfaces of the inclusions is set with the **'small_space'** parameter.:

```
|                    period                    |
  | small_space  |
|2|----------------|   1   |--------------| 2 |
```

Inclusion1 will always be centered and of material 'inclusion_a', while all higher order inclusions are made of material 'inclusion_b'.

For unit cells that contain 3 or more inclusions there are 2 implemented spacing options. By **default 'edge_spacing = False'** and the centers of all inclusions are equally spaced, with inclusion1 centered in the middle of the unit cell.

```
|       |   equally    |  seperated   |       |
|--|    2    |------|  1  |---------| 3 |-----|
```

The alternative is to space the inclusions with equal distances between their surfaces. This is selected with the keyword argument **'edge_spacing = True'**:

```
|  |       |equally |     |seperat-|   | ed   |
|--|   2   |--------|  1  |--------| 3 |------|
```

EMUstack can at present create mesh with up to 6 inclusions. It is straightforward to extended this.

5.2 2D FEM Mode Solver

5.2.1 2D Mesh

2D FEM mesh are created using the open source program gmsh. In general they are created automatically by EMUstack using the templates files for each inclusion shape. These are stored in backend/fortran/msh. For an up to date list of templates see the 'inc_shape' entry in the NanoStruct docstring.

An advantage of using the FEM to calculate the modes of layers is that there is absolutely no constraints on the content of the unit cell. If you wish to create a different structure this can be done using gmsh, which is also used to view the mesh files (select files with the extension .msh).

Note that the area of the unit cell must always be unity! This has been assumed throughout the theoretical derivations.

5.2.2 FEM Errors

There are 2 errors that can be easily triggered within the Fortran FEM routines. These both cause them to simulation to abort and the terminal to be unresponsive (until you kill python or the screen session as described in *screen_sesh*).

The first of these is

```
Error with _naupd, info_32 =           -3
Check the documentation in _naupd.
Aborting...
```

Long story short, this indicates that the FEM mesh is too coarse for solutions for higher order Bloch modes (Eigenvaules) to converge. To see this run the simulation with FEM_debug = 1 (in mode_calcs.py) and it will print the number of converged Eigenvalues nconv != nval. This error is easily fixed by increasing the mesh resolution. Decrease 'lc_bkg' and/or increase 'lc2' etc.

The second error is

```
Error with _naupd, info_32 =           -8
Check the documentation in _naupd.
Aborting...
```

This is the opposite problem, when the mesh is so fine that the simulation is overloading the memory of the machine. More accurately the memory depends on the number of Eigenvalues being calculated as well as the number of FEM mesh points. The best solution to this is to increase 'lc_bkg' and/or decrease 'lc2' etc.

CHAPTER 6

Indices and tables

- *genindex*
- *modindex*
- *search*

m

materials, 45
mode_calcs, 46

o

objects, 41

p

plotting, 49

s

stack, 48

Symbols

__getstate__() (materials.Material method), 46
__setstate__() (materials.Material method), 46

A

air_ref() (mode_calcs.Modes method), 47
Anallo (class in mode_calcs), 46

B

Bloch_fields_1d() (in module plotting), 49
BM_amplitudes() (in module plotting), 49

C

calc_1d_grating_orders() (mode_calcs.Modes method), 47
calc_2d_grating_orders() (mode_calcs.Modes method), 47
calc_kz() (mode_calcs.Anallo method), 46
calc_modes() (mode_calcs.Anallo method), 46
calc_modes() (mode_calcs.Simmo method), 47
calc_modes() (objects.NanoStruct method), 43
calc_modes() (objects.ThinFilm method), 44
calc_scat() (stack.Stack method), 48
calculate_ff() (in module objects), 44
clear_previous() (in module plotting), 51

D

dec_float_str() (in module objects), 44

E

E_conc_plot() (in module plotting), 49
EMUstack (class in mode_calcs), 47
EMUstack (class in objects), 41
EOT_plot() (in module plotting), 49
evanescent_merit() (in module plotting), 51
extinction_plot() (in module plotting), 51

F

Fabry_Perot_res() (in module plotting), 50
field_values() (in module plotting), 51
fields_3d() (in module plotting), 52
fields_in_plane() (in module plotting), 52
fields_interpolator_in_plane() (in module plotting), 52
fields_vertically() (in module plotting), 52

G

gen_params_string() (in module plotting), 53

H

heights_nm() (stack.Stack method), 48
heights_norm() (stack.Stack method), 48

J

J_sc_eta_NO_plots() (in module plotting), 50
J_short_circuit() (in module plotting), 50

K

k() (mode_calcs.Anallo method), 46
k_pll_norm() (mode_calcs.Modes method), 47

L

layers_plot() (in module plotting), 53
layers_print() (in module plotting), 53
Light (class in objects), 41

M

make_mesh() (objects.NanoStruct method), 44
Material (class in materials), 45
materials (module), 45
max_n() (in module plotting), 53
mode_calcs (module), 46
Modes (class in mode_calcs), 47

N

n() (materials.Material method), 46
n() (mode_calcs.Anallo method), 47
NanoStruct (class in objects), 41

O

objects (module), 41

omega_plot() (in module plotting), 53

P

plot_n_data() (in module materials), 46
plotting (module), 49
prop_fwd() (mode_calcs.Modes method), 47
PW_amplitudes() (in module plotting), 51

R

r_t_mat() (in module mode_calcs), 47
r_t_mat_anallo() (in module mode_calcs), 47
r_t_mat_tf_ns() (in module mode_calcs), 47

S

shear_transform() (mode_calcs.Modes method), 47
Simmo (class in mode_calcs), 47
specular_incidence() (mode_calcs.Anallo method), 47
Stack (class in stack), 48
stack (module), 48
structures() (stack.Stack method), 48

T

t_func_k_plot_1D() (in module plotting), 53
t_r_a_plots() (in module plotting), 53
t_r_a_plots_subs() (in module plotting), 54
t_r_a_write_files() (in module plotting), 54
ThinFilm (class in objects), 44
tick_function() (in module plotting), 55
total_height() (stack.Stack method), 48
total_tra_plot() (in module plotting), 55
total_tra_plot_subs() (in module plotting), 55

U

ult_efficiency() (in module plotting), 55

V

vis_matrix() (in module plotting), 55
vis_scat_mats() (in module plotting), 55

W

wl_norm() (mode_calcs.Modes method), 47

Z

Z() (mode_calcs.Anallo method), 46
zeros_int_str() (in module plotting), 55