EMUstack Documentation

Release 0.8.0

Björn Sturmberg

June 16, 2014

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 Introduction

EMUstack calculates the scattering matrices of a multi-layered structure, where each layer can be homogeneous or structured (down to sub-wavelength dimensions) and the materials may have complex, dispersive refractive indices. The scattering matrices are powerful tools from which many physical quantities, such as the total transmission, absorption in each layer, and the resonances of the structure, can be derived.

An advantage of EMUstack over other scattering matrix programs (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 therefore 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.

Inherent to the scattering matrix approach is the requirement that the interfaces between layers be planar, *ie.* that each layer is uniform in one direction (here labelled {it z}). In this nomenclature the incident field must have $k_z = k_{perp} ne 0$, but is unconstrained in $k_{parallel} = k_{x,y}$. In our implementation, the only constraint placed on each layer in the x-y plane, is that it must be periodic, at least at the supercell level. This is because the modes of structured media are calculated using a vectorial Finite Element Method (FEM) routine with periodic boundary conditions. The scattering matrices of homogeneous media are calculated analytically resulting in excellent accuracy and speed.

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 (for example MEEP). It is also possible to include media with lossless and/or non-dispersive refractive indices in EMUstack.

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.

By integrating a 2D finite element method calculation into the scattering matrix method EMUstack provides a powerful, versatile tool for nanophotonic simulations that are both computationally efficient and physically insightful.

EMUstack is a completely open source package, utilising free, open source compilers, meshing programs and libraries! The low-level numerical routines are written in Fortran for optimal performance, while higher-level processing is done in python. EMUstack currently comes with template FEM mesh for 1D and 2D gratings, such as lamellar gratings and Nanowire/Nanohole arrays. For these structures the EMUstack will automatically create FEM mesh with the specified parameters. For other structures, the open source program gmsh may be used to create the FEM mesh.

In summary, the advantages of EMUstack are;

- Calculates the scattering matrices between layers in their natural bases, for maximum physical insights.
- Designed to include lossy, dispersive materials, with frequency resolved (experimentally measured) refractive indices.
- FEM allows for arbitrary in-plane geometries, down to the periodicity of the supercell.
- Homogeneous layers are calculated analytically for optimal accuracy and speed.
- The scattering matrix method efficiently combines arbitrary number of layers into a stack.
- Synthesis of efficient Fortran routine with dynamic, high-level programming in Python.
- Completely open source package! Including FEM meshing program, Fortran FEM routine, Python multi-layered scattering matrix implementation.
- Integrated with highly optimised libraries (but also functions without these at slower speeds), including; BLAS, LAPACK, ARPACK, UMFPACK
- Vectorial FEM advantages?
- Get band structure at same time as t,r,a.
- Both/all polarasations at once

CHAPTER

TWO

INSTALLATION

2.1 Installation

The source code for EMUstack is hosted here on Github. The most convenient method of installing EMUstack is to download the installation script as follows (the alternative being to download direct from Github).

Open a terminal and navigate to the directory where you wish to install and run EMUstack from. Then run the following commands:

```
$ wget -O EMUstack_setup.sh \
http://www.physics.usyd.edu.au/emustack/setup_scripts/EMUstack_setup.sh

$ chmod +x EMUstack_setup.sh

$ bash EMUstack_setup.sh
```

This will download the latest release of EMUstack, install all dependencies (such as gfortran and linear algebra libraries) and compile EMUstack. Finally it will run some test simulations, after which you are all ready to go!

CHAPTER

THREE

TUTORIAL

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 tutorial 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/

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

3.1 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 stacked structure ##############
    """ Now when defining full structure order is critical and
    stack 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!
simotime = str(time.strftime("%Y%m%d%H%M%S", time.localtime()))
np.savez('Simo_results'+simotime, 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.
wl_num = -1
lay = 0
R12_sub = stacks_list[wl_num].layers[lay].T12
# For instance is the reflection scattering matrix off the top
# of the substrate when considered as an isolated layer.
print 'k_z of superstrate \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)

# ps 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()
```

3.2 Dispersion

```
"""
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()

# Remove results of previous simulations.
plotting.clear_previous('.txt')
plotting.clear_previous('.pdf')
plotting.clear_previous('.log')

################ 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 stacked structure ##############
    """ Now when defining full structure order is critical and
    stack 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!
simotime = str(time.strftime("%Y%m%d%H%M%S", time.localtime()))
np.savez('Simo_results'+simotime, 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 the
# plane 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()
```

3.3 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('.txt')
plotting.clear_previous('.pdf')
plotting.clear_previous('.log')

################ 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)
TF_1 = objects.ThinFilm(period, height_nm = 100, # specify thickness in nm
    material = materials.Material(2.0 + 0.1j)) # give it a constant refractive index
TF_2 = objects.ThinFilm(period, height_nm = 5e6, # EMUstack calc time is independent of height
    material = materials.InP, loss=False) # dispersive refractive index, but with
# the imaginary part of n set to zero for all wavelengths.
TF_3 = objects.ThinFilm(period, height_nm = 52,
    material = materials.Si_a) # by default loss = True
substrate   = objects.ThinFilm(period, height_nm = 'semi_inf',
    material = materials.Si_c, loss=False) # Crystaline silicon
# 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.

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 stacked structure ##############
    """ Now when defining full structure order is critical and
    stack 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!
simotime = str(time.strftime("%Y%m%d%H%M%S", time.localtime()))
np.savez('Simo_results'+simotime, 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()
```

3.4 1D Grating

```
"""
Example coming once 1.5D EMUstack is created...
"""

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('.txt')
plotting.clear_previous('.pdf')
plotting.clear_previous('.log')

################ 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 = 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)
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, make_mesh_now = True, force_mesh = False, lc_bkg = 0.1, lc2= 3.0)
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_grating = grating.calc_modes(light)
    sim_substrate   = substrate.calc_modes(light)
    ################ Evaluate stacked structure ##############
    """ Now when defining full structure order is critical and
    stack 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_grating, sim_superstrate))
    stack.calc_scat(pol = 'TM')

    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!
simotime = str(time.strftime("%Y%m%d%H%M%S", time.localtime()))
np.savez('Simo_results'+simotime, 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()
```

3.5 2D Grating

```
"""
Simulating NW array with period 600 nm and NW diameter 120 nm, placed ontop of
different substrates.
"""

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 im simulation
num_cores = 7

# Remove results of previous simulations
plotting.clear_previous('.txt')
plotting.clear_previous('.pdf')
plotting.clear_previous('.gif')
plotting.clear_previous('.log')

################ 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 = 600.65
max_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_a, 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)

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)

    ################ Evaluate full solar cell structure ##############
    """ Now when defining full structure order is critical and
    solar_cell 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!
simotime = str(time.strftime("%Y%m%d%H%M%S", time.localtime()))
np.savez('Simo_results'+simotime, stacks_list=stacks_list)


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

plotting.t_r_a_plots(stacks_list, active_layer_nu=1, J_sc=True)
# Dispersion
plotting.omega_plot(stacks_list, wavelengths)


#Accessing scattering matrices of individual layers, and interfaces.
# betas = stacks_list[0][0][0].layers[1].k_z
# print betas
# betas = stacks_list[0][0][0].layers[0].k_z
# print betas
```

```
# Rnet = stacks_list[0][0][0].R_net
# J_mat = stacks_list[0][0][0].layers[1].J
# T_c = np.sum((np.abs(stacks_list[0][0][0].layers[1].T12)), axis=1)
# print T_c
# print Rnet
# print J_mat
# print_fmt = zip(np.real(betas),np.imag(betas),T_c)
# np.savetxt('Coupling_beta.txt', print_fmt, fmt = '%7.4f')
# print_fmt = zip(np.real(betas),np.imag(betas))
# np.savetxt('Coupling_beta.txt', print_fmt)



######################## 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 ''
```

3.6 Angles of Incidence

```
"""
Template python script file to execute a simulation. To start, open a terminal and change
directory to the directory containing this file (which must be in the same directory as
the EMUstack directory). Run this script file by executing the following in the command line

$ python simmo_NW_array.py

This will use num_cores worth of your CPUs, and by default return you in the command
line, having printed results and saved plots to file as specified towards the end of
this file. If instead you wish to have direct access to the simulation results (for
further manipulation, debugging etc.) run this script with

$ python -i simmo_NW_array.py

which, after the calculations are complete, will return you into an interactive session
of python, in which all simulation objects are accessible. In this session you can access
the docstrings of objects/classes/methods by typing

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

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

```
In real simulation scripts replace this docstring with a brief description of the
simulation, eg.
'Simulating NW array with period 600 nm and NW diameter 120 nm, placed ontop of
different substrates.'
"""

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 im simulation
num_cores = 7

# Remove results of previous simulations
plotting.clear_previous('.txt')
plotting.clear_previous('.pdf')
plotting.clear_previous('.gif')
plotting.clear_previous('.log')

################ 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 = 3) for wl in wavelengths]


# period must be consistent throughout simulation!!!
period = 600
max_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_a, 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)

# Find num_BM for each simulation (wl) as num decreases w decreasing index contrast.
max_n = max([NW_array.inclusion_a.n(wl).real for wl in wavelengths])

def simulate_stack(light):
```

```
    num_BM = round(max_num_BMs * NW_array.inclusion_a.n(light.wl_nm).real/max_n)
    # num_BM = max_num_BMs

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

    ################ Evaluate full solar cell structure ##############
    """ Now when defining full structure order is critical and
    solar_cell 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!
simotime = str(time.strftime("%Y%m%d%H%M%S", time.localtime()))
np.savez('Simo_results'+simotime, stacks_list=stacks_list)


######################## Plotting ########################
#### Example 1: simple multilayered stack.
plotting.t_r_a_plots(stacks_list)
# Dispersion
plotting.omega_plot(stacks_list, wavelengths, stack_label=stack_label)


######################## 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 ''
```

3.7 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 im simulation
num_cores = 15

# Remove results of previous simulations
plotting.clear_previous('.txt')
plotting.clear_previous('.pdf')
plotting.clear_previous('.gif')
plotting.clear_previous('.log')

################ 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)

Au_NHs = objects.NanoStruct('2D_array', period, diam1,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)

num_BM = 50

def simulate_stack(light):
    num_h = 21
    NH_heights = [60]#np.linspace(10,100,num_h)

    ################ Evaluate each layer individually ##############
```

```
    sim_superstrate = superstrate.calc_modes(light)
    sim_Au    = Au_NHs.calc_modes(light, num_BM = num_BM)
    sim_substrate = substrate.calc_modes(light)

# Loop over heights
    height_list = []
    # for h in NH_heights:
    stackSub = Stack((sim_substrate, sim_Au, sim_superstrate))#, heights_nm = ([h]))
    stackSub.calc_scat(pol = 'R Circ')
    stackSub2 = Stack((sim_substrate, sim_Au, sim_superstrate))#, heights_nm = ([h]))
    stackSub2.calc_scat(pol = 'L Circ')
    saveStack = Stack((sim_substrate, sim_Au, sim_superstrate))#, heights_nm = ([h]))

    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
    height_list.append(saveStack)
    # 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!
simotime = str(time.strftime("%Y%m%d%H%M%S", time.localtime()))
np.savez('Simo_results'+simotime, stacks_list=stacks_list)

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


#### Individual spectra of multilayered stack where one layer has many heights.
stack_label = 0
active_layer_nu = 1
# for h in range(num_h):
h = 0
gen_name = '_h-'
h_name = str(h)
additional_name = gen_name+h_name # You can add an arbitry string onto the end of the spectra filenar
stack3_hs_wl_list = []
for i in range(len(wavelengths)):
    stack3_hs_wl_list.append(stacks_list[i][h])
plotting.t_r_a_plots(stack3_hs_wl_list, stack_label=stack_label, add_name = additional_name)


# 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 ''
```

3.8 Plotting Fields

```
"""
Template python script file to execute a simulation. To start, open a terminal and change
directory to the directory containing this file (which must be in the same directory as
the EMUstack directory). Run this script file by executing the following in the command line

$ python simmo_NW_array.py

This will use num_cores worth of your CPUs, and by default return you in the command
line, having printed results and saved plots to file as specified towards the end of
this file. If instead you wish to have direct access to the simulation results (for
further manipulation, debugging etc.) run this script with

$ python -i simmo_NW_array.py

which, after the calculations are complete, will return you into an interactive session
of python, in which all simulation objects are accessible. In this session you can access
the docstrings of objects/classes/methods by typing

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

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


In real simulation scripts replace this docstring with a brief description of the
simulation, eg.
'Simulating NW array with period 600 nm and NW diameter 120 nm, placed ontop of
different substrates.'
"""

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 im simulation
num_cores = 7

# Remove results of previous simulations
plotting.clear_previous('.txt')
plotting.clear_previous('.pdf')
plotting.clear_previous('.gif')
plotting.clear_previous('.log')

################ 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 = 3) for wl in wavelengths]


# period must be consistent throughout simulation!!!
period = 600
max_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_a, 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)

# Find num_BM for each simulation (wl) as num decreases w decreasing index contrast.
max_n = max([NW_array.inclusion_a.n(wl).real for wl in wavelengths])

def simulate_stack(light):
    num_BM = round(max_num_BMs * NW_array.inclusion_a.n(light.wl_nm).real/max_n)
    # num_BM = max_num_BMs

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

    ################ Evaluate full solar cell structure ##############
    """ Now when defining full structure order is critical and
    solar_cell 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!
simotime = str(time.strftime("%Y%m%d%H%M%S", time.localtime()))
np.savez('Simo_results'+simotime, stacks_list=stacks_list)


######################## Plotting ########################
#### Example 1: simple multilayered stack.

plotting.t_r_a_plots(stack_wl_list)

# Dispersion
plotting.omega_plot(stack_wl_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 ''
```

3.9 Plotting Amplitudes

```
"""

"""

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 im simulation
num_cores = 5

# Remove results of previous simulations
plotting.clear_previous('.txt')
plotting.clear_previous('.pdf')
plotting.clear_previous('.log')

################ Light parameters ####################
azi_angles = np.linspace(0,89,5)
wl = 1600
light_list  = [objects.Light(wl, max_order_PWs = 4, theta = p, phi = 0.0) for p in azi_angles]


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

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)

absorber    = objects.ThinFilm(period, height_nm = 10,
    material = materials.Material(2.0 + 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, make_mesh_now = True, force_mesh = False, lc_bkg = 0.1, lc2= 3.0)


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 full solar cell structure ##############
    """ Now when defining full structure order is critical and
    stack 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!
```

```
simotime = str(time.strftime("%Y%m%d%H%M%S", time.localtime()))
np.savez('Simo_results'+simotime, stacks_list=stacks_list)

######################## Post Processing ########################
# We can plot the amplitudes of each transmitted plane wave order as a function of angle.
plotting.amps_of_orders(stacks_list, add_title='-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.
plotting.amps_of_orders(stacks_list, 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)

# We can represent the strength with which different orders are excited in k-space.
plotting.t_func_k_plot_1D(stacks_list)
# This corresponds to Fig 2 of Handmer et al. Optics Lett. 35, 2010.
# (The amps_of_orders plots correspond to Fig 1 of this paper).

# Lastly we also plot the transmission, reflection and absorption of each layer and the stack.
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 ''
```

3.10 Shear Transformations

```
"""

"""

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 im simulation
num_cores = 5

# Remove results of previous simulations
plotting.clear_previous('.txt')
plotting.clear_previous('.pdf')
plotting.clear_previous('.log')

################ 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_BM = 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_BM = num_BM)
    sim_grating_2   = grating_2.calc_modes(light, num_BM = num_BM)

    ################ Evaluate full solar cell structure ##############
    """ Now when defining full structure order is critical and
    stack 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!
simotime = str(time.strftime("%Y%m%d%H%M%S", time.localtime()))
np.savez('Simo_results'+simotime, 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 ''
```

3.11 Varying a Single Layer

```
"""
Simulating solar cell efficiency of nanohole array of set geometry
as vary substrate and superstrate refractive indeces.
CAUTION: very memory intensive!
"""

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 im simulation
num_cores = 7

# Remove results of previous simulations
```

```
plotting.clear_previous('.txt')
plotting.clear_previous('.pdf')
plotting.clear_previous('.gif')
plotting.clear_previous('.log')

################ 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 = 3) for wl in wavelengths]
# Single wavelength run
# wl_super = 450
# wavelengths = np.array([wl_super])
# light_list  = [objects.Light(wl) for wl in wavelengths]


# period must be consistent throughout simulation!!!
period = 600
max_num_BMs = 200

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

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

ThinFilm2  = objects.ThinFilm(period, height_nm = 100,
    material = materials.SiO2_a, loss = True)

ThinFilm4  = objects.ThinFilm(period, height_nm = 200,
    material = materials.Si_c, loss = True)

NW_diameter = 120
NWs = 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 = 0.2, lc2= 1.0)

# Find num_BM for each simulation (wl) as num decreases w decreasing index contrast.
max_n = max([NWs.inclusion_a.n(wl).real for wl in wavelengths])

def simulate_stack(light):
    num_BM = round(max_num_BMs * NWs.inclusion_a.n(light.wl_nm).real/max_n)
    # num_BM = max_num_BMs

    ################ Evaluate each layer individually ##############
    sim_superstrate = superstrate.calc_modes(light)
    sim_substrate   = substrate.calc_modes(light)
    sim_ThinFilm2   = ThinFilm2.calc_modes(light)+
    sim_ThinFilm4   = ThinFilm4.calc_modes(light)
    sim_NWs         = NWs.calc_modes(light, num_BM = num_BM)

    ################ Evaluate full solar cell structure ##############
    """ Now when defining full structure order is critical and
    solar_cell list MUST be ordered from bottom to top!
    """
```

```
    stack0 = Stack((sim_substrate, sim_superstrate))
    stack1 = Stack((sim_substrate, sim_NWs, sim_superstrate))
    # stack1 = Stack((sim_substrate, sim_ThinFilm2, sim_NWs, sim_superstrate))
    # stack1 = Stack((sim_substrate, sim_ThinFilm2, sim_ThinFilm4 sim_superstrate))
    stack0.calc_scat(pol = 'TE')
    stack1.calc_scat(pol = 'TE')

# multiple heights for sim_ThinFilm4
    stack2_indiv_hs = []
    average_t = 0
    average_r = 0
    average_a = 0

    num_h = 10
    for h in np.linspace(100,2000,num_h):
        stack2 = Stack((sim_substrate,sim_ThinFilm2,sim_ThinFilm4 sim_superstrate),
         heights_nm = ([sim_ThinFilm2.height_nm,h]))
        stack2.calc_scat(pol = 'TE')

        stack2_indiv_hs.append(stack2)

        average_t += stack2.t_list[-1]/num_h
        average_r += stack2.r_list[-1]/num_h
        average_a += stack2.a_list[-1]/num_h
    stack2.t_list[-1] = average_t
    stack2.r_list[-1] = average_r
    stack2.a_list[-1] = average_a

# stack2 contains info on the last height and the average spectra
    return [stack0,stack1,stack2,stack2_indiv_hs]
    # return stack1


# 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!
simotime = str(time.strftime("%Y%m%d%H%M%S", time.localtime()))
np.savez('Simo_results'+simotime, stacks_list=stacks_list)




# Pull apart simultaneously simulated stakes into single stack, all wls arrays.
# Unnecissary if just returning a single stack
np.array(stacks_list)


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




param_layer = NWs # Specify the layer for which the parameters should be printed on figures.
params_string = plotting.gen_params_string(param_layer, last_light_object, max_num_BMs=max_num_BMs)
```

```
#### Example 1: simple multilayered stack.
stack_label = 1 # 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])
active_layer_nu = 1

Efficiency = plotting.t_r_a_plots(stack1_wl_list, wavelengths, params_string,
    active_layer_nu=active_layer_nu, stack_label=stack_label)
# Dispersion
plotting.omega_plot(stack1_wl_list, wavelengths, params_string, stack_label=stack_label)
```

```
# #### Example 0: simple interface.
# param_layer = bottom1 # Specify the layer for which the parameters should be printed on figures.
# params_string = plotting.gen_params_string(param_layer, last_light_object)
# stack_label = 0 # Specify which stack you are dealing with.
# stack0_wl_list = []
# for i in range(len(wavelengths)):
#     stack0_wl_list.append(stacks_list[i][stack_label])
# # Plot total transmission, reflection, absorption & that of each layer.
# Efficiency = plotting.t_r_a_plots(stack0_wl_list, wavelengths, params_string,
#     stack_label=stack_label)
```

```
# param_layer = TF4 # Specify the layer for which the parameters should be printed on figures.
# params_string = plotting.gen_params_string(param_layer, last_light_object, max_num_BMs=max_num_BMs,

# #### Example 1: simple multilayered stack.
# stack_label = 1 # 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])
# active_layer_nu = 2 # Specify which layer is the active one (where absorption generates charge carr
# # Plot total transmission, reflection, absorption & that of each layer.
# # Also calculate efficiency of active layer.
# Efficiency = plotting.t_r_a_plots(stack1_wl_list, wavelengths, params_string,
#     active_layer_nu=active_layer_nu, stack_label=stack_label)
# # Dispersion
# plotting.omega_plot(stack1_wl_list, wavelengths, params_string, stack_label=stack_label)
# # # Energy Concentration
# # which_layer = 2
# # which_modes = [1,2] # can be a single mode or multiple modes
# # plotting.E_conc_plot(stack1_wl_list, which_layer, which_modes, wavelengths,
```

```
# #     params_string, stack_label=stack_label)



# #### Example 2: averaged multilayered stack where one layer has many heights.
# stack_label = 2
# active_layer_nu = 2
# stack2_wl_list = []
# for i in range(len(wavelengths)):
#     stack2_wl_list.append(stacks_list[i][stack_label])
# Efficiency = plotting.t_r_a_plots(stack2_wl_list, wavelengths, params_string,
#     active_layer_nu=active_layer_nu, stack_label=stack_label)



# #### Example 3: individual spectra of multilayered stack where one layer has many heights.
# stack_label = 3
# active_layer_nu = 2
# number_of_hs = len(stacks_list[0][stack_label])
# for h in range(number_of_hs):
#     gen_name = '_h-'
#     h_name = str(h)
#     additional_name = gen_name+h_name # You can add an arbitry string onto the end of the spectra
#     stack3_hs_wl_list = []
#     for i in range(len(wavelengths)):
#         stack3_hs_wl_list.append(stacks_list[i][stack_label][h])
#     Efficiency = plotting.t_r_a_plots(stack3_hs_wl_list, wavelengths, params_string,
#         active_layer_nu=active_layer_nu, stack_label=stack_label, add_name = additional_name)
# # Animate spectra as a function of heights.
# from os import system as ossys
# delay = 5 # delay between images in gif in hundredths of a second
# names = 'Lay_Absorb_stack'+str(stack_label)+gen_name
# 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)



######################## Single Wavelength Plotting ########################
# Plot transmission as a function of k vector.
# plotting.t_func_k_plot(stack3_wl_list)

# # Visualise the Scattering Matrices
# for i in range(len(wavelengths)):
#     extra_title = 'R_net'
#     plotting.vis_scat_mats(stack1_wl_list[i].R_net, i, extra_title)
#     extra_title = 'R_12'
#     plotting.vis_scat_mats(stack1_wl_list[i].layers[2].T21, i, extra_title)
```

```
######################## Wrapping up ########################
print '\n*****************************************'
print 'The ultimate efficiency is %12.8f' % Efficiency
print '-----------------------------------------'

# 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 Convergence Testing

```
# Number of CPUs to use im simulation
num_cores = 8

# Remove results of previous simulations
plotting.clear_previous('.log')
plotting.clear_previous('.txt')
plotting.clear_previous('.pdf')
plotting.clear_previous('.gif')

################ 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_BM = 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_BM = num_BM)
        sim_grating_2   = grating_2.calc_modes(light, num_BM = num_BM)

        ################ Evaluate full solar cell structure ##############
        """ Now when defining full structure order is critical and
        solar_cell 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!
    simotime = str(time.strftime("%Y%m%d%H%M%S", time.localtime()))
    np.savez('Simo_results'+simotime, 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 ''
```

3.13 Stacked Gratings with Angles

```
# Number of CPUs to use im simulation
num_cores = 8

# Remove results of previous simulations
plotting.clear_previous('.log')
plotting.clear_previous('.pdf')

################ Light parameters ####################
# wavelengths = np.linspace(800,1600,100)
# light_list  = [objects.Light(wl, max_order_PWs = 6, theta = 0.0, phi = 0.0) for wl in wavelengths]
wl = 1600
azi_angles = np.linspace(0,89,90)
light_list  = [objects.Light(wl, max_order_PWs = 6, 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 = 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_BM = 250


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_BM = num_BM)
    sim_grating_2   = grating_2.calc_modes(light, num_BM = num_BM)

    ################ Evaluate full solar cell structure ##############
    """ Now when defining full structure order is critical and
    solar_cell 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!
simotime = str(time.strftime("%Y%m%d%H%M%S", time.localtime()))
np.savez('Simo_results'+simotime, stacks_list=stacks_list)


# ### Plot as in Handmer Fig2
plotting.t_func_k_plot_1D(stacks_list)

### Plot as in Handmer Fig1
plotting.single_order_T(stacks_list)

plotting.t_r_a_plots(stack_wl_list)

# select_stack = stacks_list[-1]
# plot_mat = select_stack.T_net
# num_prop_PWs = select_stack.layers[0].num_prop_pw_per_pol
# plotting.vis_scat_mats(plot_mat,num_prop_PWs,extra_title='Transmission')

######################## 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 ''
```

3.14 Stacked Gratings with Wavelengths

```
# Number of CPUs to use im simulation
num_cores = 8

# Remove results of previous simulations
plotting.clear_previous('.log')
plotting.clear_previous('.pdf')

################ Light parameters ####################
wavelengths = np.linspace(800,1600,101)
light_list  = [objects.Light(wl, max_order_PWs = 6, theta = 0.0, phi = 0.0) for wl in wavelengths]
# azi_angles = np.linspace(45,55,30)
# light_list  = [objects.Light(wl, max_order_PWs = 6, 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 = 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_BM = 250


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_BM = num_BM)
    sim_grating_2   = grating_2.calc_modes(light, num_BM = num_BM)

    ################ Evaluate full solar cell structure ##############
    """ Now when defining full structure order is critical and
    solar_cell 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!
simotime = str(time.strftime("%Y%m%d%H%M%S", time.localtime()))
np.savez('Simo_results'+simotime, stacks_list=stacks_list)


# ### Plot as in Handmer Fig2
# # require phi == 0.0
last_light_object = light_list.pop()
# n_H = 3.61 # high refractive index
# min_k_label = 15
# plotting.t_func_k_plot_1D(stacks_list, last_light_object, n_H, min_k_label)


# ### Plot as in Handmer Fig1
# chosen_PW_order = [-1,0,1,2]
# plotting.single_order_T(stacks_list, azi_angles, chosen_PW_order)
```

```
stack_wl_list = []
for i in range(len(wavelengths)):
    stack_wl_list.append(stacks_list[i])
active_layer_nu = 1


plotting.t_r_a_plots(stack_wl_list, active_layer_nu=active_layer_nu)


# select_stack = stacks_list[-1]
# plot_mat = select_stack.T_net
# num_prop_PWs = select_stack.layers[0].num_prop_pw_per_pol
# plotting.vis_scat_mats(plot_mat,num_prop_PWs,extra_title='Transmission')

######################## 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 ''
```

3.15 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 im simulation
num_cores = 16

# Remove results of previous simulations
plotting.clear_previous('.txt')
plotting.clear_previous('.pdf')
plotting.clear_previous('.gif')
plotting.clear_previous('.log')

################ 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 = 4) 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,
    square = True,
    make_mesh_now = True, force_mesh = True, lc_bkg = 0.12, lc2= 5.0, lc3= 3.0)#lc_bkg = 0.08, lc2= 3

superstrate = objects.ThinFilm(period = period, height_nm = 'semi_inf',
    material = materials.Air, loss = False)
substrate   = 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_superstrate  = superstrate.calc_modes(light)
    sim_substrate    = substrate.calc_modes(light)

# Loop over heights
    height_list = []
    for h in NH_heights:
        stackSub = Stack((sim_substrate, sim_NHs, sim_superstrate), 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!
simotime = str(time.strftime("%Y%m%d%H%M%S", time.localtime()))
```

```
np.savez('Simo_results'+simotime, 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)
# 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 ''
```

3.16 Resonant Grating

```
"""
Template python script file to execute a simulation. To start, open a terminal and change
directory to the directory containing this file (which must be in the same directory as
the EMUstack directory). Run this script file by executing the following in the command line

$ python simmo_resonant_grating.py

This will use num_cores worth of your CPUs, and by default return you in the command
line, having printed results and saved plots to file as specified towards the end of
this file. If instead you wish to have direct access to the simulation results (for
further manipulation, debugging etc.) run this script with

$ python -i simmo_resonant_grating.py

which, after the calculations are complete, will return you into an interactive session
of python, in which all simulation objects are accessible. In this session you can access
```

```
the docstrings of objects/classes/methods by typing

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

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


In real simulation scripts replace this docstring with a brief description of the
simulation, eg.
'Simulating the coupling of normally incident light into evanescent orders through a
metallic grating of period 120 nm. Included 3 PW 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 im simulation
num_cores = 5

# Remove results of previous simulations
plotting.clear_previous('.txt')
plotting.clear_previous('.pdf')
plotting.clear_previous('.gif')
plotting.clear_previous('.log')

################ Light parameters ####################
wl_1     = 900
wl_2     = 1200
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 = 3) for wl in wavelengths]


# period must be consistent throughout simulation!!!
period = 120
num_BM = 90

superstrate = objects.ThinFilm(period, height_nm = 'semi_inf',
    material = materials.Material(3.5 + 0.0j), loss = True)

homo_film  = objects.ThinFilm(period, height_nm = 5,
    material = materials.Material(3.6 + 0.27j), loss = True)
```

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

grating_diameter = 100
grating = objects.NanoStruct('1D_array', period, grating_diameter, height_nm = 25,
    inclusion_a = materials.Ag, background = materials.Material(1.5 + 0.0j), loss = True,
    make_mesh_now = True, force_mesh = True, lc_bkg = 0.05, lc2= 4.0)

mirror = objects.ThinFilm(period, height_nm = 100,
    material = materials.Ag, loss = True)


def simulate_stack(light):
    ################ Evaluate each layer individually ##############
    sim_superstrate = superstrate.calc_modes(light)
    sim_homo_film   = homo_film.calc_modes(light)
    sim_substrate   = substrate.calc_modes(light)
    sim_grating     = grating.calc_modes(light, num_BM = num_BM)
    sim_mirror      = mirror.calc_modes(light)

    ################ Evaluate full solar cell structure ##############
    """ Now when defining full structure order is critical and
    solar_cell list MUST be ordered from bottom to top!
    """
    stack = Stack((sim_substrate, sim_mirror, sim_grating, sim_homo_film, 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!
simotime = str(time.strftime("%Y%m%d%H%M%S", time.localtime()))
np.savez('Simo_results'+simotime, stacks_list=stacks_list)


######################## Plotting ########################
last_light_object = light_list.pop()
param_layer = grating_1 # Specify the layer for which the parameters should be printed on figures.
params_string = plotting.gen_params_string(param_layer, last_light_object, max_num_BMs=num_BM)


active_layer_nu = 3 # Specify which layer is the active one (where absorption generates charge carrie
# Plot total transmission, reflection, absorption & that of each layer.
# Also calculate efficiency of active layer.
Efficiency = plotting.t_r_a_plots(stacks_list, wavelengths, params_string, active_layer_nu=active_lay
# Dispersion
# plotting.omega_plot(stacks_list, wavelengths, params_string)
# # Energy Concentration
# which_layer = 2
# which_modes = [1,2] # can be a single mode or multiple
# plotting.E_conc_plot(stacks_list, which_layer, which_modes, wavelengths,
#     params_string)
```

```
######################## Single Wavelength Plotting ########################
# Plot transmission as a function of k vector.
# plotting.t_func_k_plot(stacks_list)


# # Visualise the Scattering Matrices
# for i in range(len(wavelengths)):
#     extra_title = 'R_net'
#     plotting.vis_scat_mats(stacks_list[i].R_net, i, extra_title)
#     extra_title = 'R_12'
#     plotting.vis_scat_mats(stacks_list[i].layers[2].T21, i, extra_title)







# betas = stacks_wl_list[0][0][0].layers[1].k_z
# print betas
# betas = stacks_wl_list[0][0][0].layers[0].k_z
# print betas

# Rnet = stacks_wl_list[0][0][0].R_net
# J_mat = stacks_wl_list[0][0][0].layers[1].J
# T_c = np.sum((np.abs(stacks_wl_list[0][0][0].layers[1].T12)), axis=1)
# print T_c
# print Rnet
# print J_mat
# print_fmt = zip(np.real(betas),np.imag(betas),T_c)
# np.savetxt('Coupling_beta.txt', print_fmt, fmt = '%7.4f')
# print_fmt = zip(np.real(betas),np.imag(betas))
# np.savetxt('Coupling_beta.txt', print_fmt)







# Wrapping up simulation by printing to screen and log file
print '\n*****************************************'
print 'The ultimate efficiency is %12.8f' % Efficiency
print '-----------------------------------------'

# 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 ''
```

3.17 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

FOUR

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.**Light**(*wl_nm, max_order_PWs=2, k_parallel=[0.0, 0.0], 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.

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.

class objects.**NanoStruct**(*geometry, period, diameter1, inc_shape='circle', ellipticity=0.0, ff=0, ff_rand=False, small_d=0, inclusion_a=<materials.Material object at 0x2ef1c90>, inclusion_b=<materials.Material object at 0x2ef1c50>, background=<materials.Material object at 0x2ef1c10>, loss=True, height_nm=1000, 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, hyperbolic=False, posx=0, posy=0, make_mesh_now=True, force_mesh=False, mesh_file='NEED_FILE.mail', lc_bkg=0.09, lc2=1.0, lc3=1.0, lc4=1.0, lc5=1.0, lc6=1.0, plot_modes=False, plot_real=1, plot_imag=0, plot_abs=0, plotting3d=False, plot_field_conc=False*)

Bases: object

Represents a structured layer.

Parameters

- **geometry** (*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

- **inc_shape** (*str*) – Shape of inclusions that have template mesh, currently; 'circle', 'ellipse', 'square', 'split ring'.
- **ellipticity** (*float*) – If != 0, inclusion has given ellipticity, with b = diameter, a = diameter-ellipticity * diameter. NOTE: only implemented for 1 inclusion.
- **diameter2-16 (float): The diameters of further inclusions in nm.**
- **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.
- **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.
- **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_d** (*float*) – Distance between 2 inclusions of interleaved 1D grating.
- **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 (eg. 600_60.mail), it must be located in backend/fem_2d/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.**
- **plot_modes** (*bool*) – Plot modes (ie. FEM solutions) in gmsh format, you get epsilon*|E|^2 & either real/imag/abs of x,y,z components, field vectors.
- **plot_real** (*bool*) – Plot the real part of modal fields.
- **plot_imag** (*bool*) – Plot the imaginary part of modal fields.

- **plot_abs** (*bool*) – Plot the absolute value of modal fields.
- **plotting3d** (*bool*) – Plot the fields in 3D.

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*, *world_1d=False*, *height_nm=1000*, *num_pw_per_pol=0*, *material=<materials.Material object at 0x2ef1b90>*, *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

- **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.
- **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.
- **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*, *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.
- **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.

__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.

class materials.**UnivariateSpline**
Bases: object

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()

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.**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_BM=None*, *delete_working=True*)
Run a Fortran FEM caluculation 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. Thisoverrides any heights specified in the :ThinFilm: or :NanoStruct: objects.

calc_scat(*pol='TE'*, *incoming_amplitudes=None*)
Calculate the transmission and reflection matrices of the stack.

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

heights_nm()

heights_norm()

shears()

structures()

total_height()

4.5 plotting module

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

plotting.**EOT_plot**(*stacks_list*, *wavelengths*, *params_layer=1*, *num_pw_per_pol=0*, *add_name=''*)
Plot T_{00} as in Martin-Moreno PRL 86 2001. To plot {9,0} component of TM polarisation set num_pw_per_pol = num_pw_per_pol.

plotting.**E_PW_fields**(*stack*, *nu_calc_pts=51*, *max_height=3*, *nu_slices=5*, *Substrate=True*, *Superstrate=True*)

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=0*, *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 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.**amps_of_orders**(*stacks_list*, *xvalues=None*, *chosen_PW_order=None*, *lay_interest=0*, *add_height=None*, *add_title=None*)
Plot the amplitudes of plane wave orders in selected layer.

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_PW_order** (*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_title** (*str*) – Add add_name to title.

plotting.**clear_previous**(*file_type*)
Delete all files of specified type 'file_type'.

plotting.**evanescent_merit**(*stacks_list*, *xvalues=None*, *chosen_PW_order=None*, *lay_interest=0*, *add_height=None*, *add_title=None*)

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_PW_order** (*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_title** (*str*) – Add add_name to title.

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

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

plotting.**fields_2d**(*pstack*, *Struc_lay=1*, *TF_lay=0*)

plotting.**fields_3d**(*pstack*, *wl*)

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*, *force_txt_save*)

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.

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*, *params_layer=1*, *active_layer_nu=0*, *stack_label=1*, *ult_eta=False*, *J_sc=False*, *weight_spec=False*, *extinct=False*, *add_height=0.0*, *add_name=''*, *force_txt_save=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 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.
- **force_txt_save** (*bool*) – If True, save spectra data to text files.

plotting.**t_r_a_plots_subs**(*stacks_list*, *wavelengths*, *period*, *sub_n*, *params_layer=1*, *active_layer_nu=0*, *stack_label=1*, *ult_eta=False*, *J_sc=False*, *weight_spec=False*, *extinct=False*, *add_height=0*, *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 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.

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*)
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*, *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).

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

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

Keyword Arguments

- **nu_prop_PWs** (*int*) – Number of propagating PWs.
- **wl** (*int*) – Index in case of calling in a loop.
- **extra_title** (*str*) – Add extra_title to title.

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

CHAPTER

FIVE

FEM BACKEND

5.1 fem_2d package

CHAPTER

SIX

INDICES AND TABLES

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

f

m

o

p

s

Symbols

A

C

D

E

F

G

H

J

K

L

M

N

O

P

R

S

T

U

V

W

Z