NumBAT Documentation

Release 0.1

Bjorn Sturmberg

February 16, 2017

Contents:

CHAPTER

ONE

INTRODUCTION

Introduction

NumBAT, the Numerical Brillouin Analysis Tool, integrates electromagnetic and acoustic mode solvers to calculate the interactions of optical and acoustic waves in waveguides.

NumBAT was developed by Bjorn Sturmberg, Kokou Dossou, Christian Wolff, Chris Poulton and Michael Steel in a collaboration between Macquarie University and the University of Technology Sydney, as part of the Australian Research Council Discovery Project DP130100832.

CHAPTER

TWO

INSTALLATION

Installation

The source code for NumBAT is hosted here on Gitlab. Please download the latest release from here.

NumBAT has been developed on Ubuntu and is easiest to install on this platform. Simply run the setup script

```
$ sudo /setup.sh
```

Or, if you prefer to do things manually, this is equivalent to

```
$ sudo apt-get update
$ sudo apt-get upgrade
$ sudo apt-get install -y <dependencies>
$ cd backend/fortran/
$ make
$ cd ../../tests/
$ nosetests
```

where the <dependencies> packages are listed dependencies.txt.

This is all there is, there isn't any more.

Well there's more if you want to change it up.

The Fortran components (NumBAT source code and libraries) have been successfully compiled with intel's ifortran as well as open-source gfortran. In this documentation we use gfortran, but this can be easily adjusted in NumBAT/backend/fortran/Makefile

On non-ubuntu OS you may also need to compile a local version of Suitesparse, which is described in the next section.

Manual installation of SuiteSparse

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

This is the process I followed in my installations, however this was some years ago and may need to be modified.

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

```
$ mkdir SS_installed/
```

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

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

set path to install folder:

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

line 290ish commenting out all other references to these:

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

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

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

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

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

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

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

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

```
$ make
```

Now move back to NumBAT/backend/fortran/

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

and then move to SuiteSparse/ and execute the following:

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

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

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

NumBAT Makefile

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

Then finally run the setup.sh script!

CHAPTER

THREE

GUIDE

Simulation Structure

Simulations with NumBAT are generally carried out using a python script file. This file is kept in its own directory which is placed in the NumBAT 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 this chapter we go through a number of example simo.py files. But before we do, another quick tip about running simulations within screen sessions, which allow you to disconnect from servers leaving them to continue your processes.

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.

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 NumBAT/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 NumBAT 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
```

Terminating NumBAT simos

If 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, or if it affects multiple instances, you can kill screen. 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
```

Basic SBS Gain Calculation

```
print(end - start)
""" Calculate the backward SBS gain for modes in a
    silicon waveguide surrounded in air.
"""

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

import materials
import objects
import mode_calcs
import integration
import plotting
from fortran import NumBAT

# Naming conventions
# AC: acoustic
# EM: electromagnetic
# k_AC: acoustic wavenumber

# Geometric Parameters - all in nm.
wl_nm = 1550 # Wavelength of EM wave in vacuum.
# Unit cell must be large to ensure fields are zero at boundary.
unitcell_x = 2.5*wl_nm
unitcell_y = unitcell_x
# Waveguide width (x direction).
inc_a_x = 314.7
# Waveguide height (y direction).
inc_a_y = 0.9*inc_a_x
# Shape of the waveguide could also be 'circular'.
inc_shape = 'rectangular'

# Optical Parameters
# Permittivity
eps = 12.25
# Number of electromagnetic modes to solve for.
num_EM_modes = 20
# Number of acoustic modes to solve for.
num_AC_modes = 20
# The first EM mode(s) for which to calculate interaction with AC modes.
# Can specify a mode number (zero has lowest propagation constant) or 'All'.
EM_ival1=0
# The second EM mode(s) for which to calculate interaction with AC modes.
EM_ival2=EM_ival1
# The AC mode(s) for which to calculate interaction with EM modes.
AC_ival='All'

# Acoustic Parameters
# Density
s = 2330  # kg/m3
# Stiffness tensor components.
c_11 = 165.7e9; c_12 = 63.9e9; c_44 = 79.6e9  # Pa
# Photoelastic tensor components
```

```
p_11 = -0.094; p_12 = 0.017; p_44 = -0.051
# Acoustic loss tensor components.
eta_11 = 5.9e-3 ; eta_12 = 5.16e-3 ; eta_44 = 0.620e-3  # Pa
# Put acoustic parameters together for convenience.
inc_a_AC_props = [s, c_11, c_12, c_44, p_11, p_12, p_44,
                  eta_11, eta_12, eta_44]

start = time.time()

# Use all specified parameters to create a waveguide object.
wguide = objects.Struct(unitcell_x,inc_a_x,unitcell_y,inc_a_y,inc_shape,
                        bkg_material=materials.Material(1.0 + 0.0j),
                        inc_a_material=materials.Material(np.sqrt(eps)),
                        loss=False, inc_a_AC=inc_a_AC_props, plotting_fields=False,
                        lc_bkg=2, lc2=2000.0, lc3=10.0)

# Expected effective index of fundamental guided mode.
n_eff = np.real(np.sqrt(eps))-0.1

# Calculate Electromagnetic Modes
sim_EM_wguide = wguide.calc_EM_modes(wl_nm, num_EM_modes, n_eff=n_eff)
# Print the wavevectors of EM modes.
print 'k_z of EM modes \n', np.round(np.real(sim_EM_wguide.Eig_values),4)
# Plot the EM modes fields, important to specify this with EM_AC='EM'.
# Zoom in on the central region (of big unitcell) with xlim, ylim args.
plotting.plt_mode_fields(sim_EM_wguide, xlim=0.4, ylim=0.4, EM_AC='EM')

# Calculate the EM effective index of the waveguide.
n_eff_sim = np.real(sim_EM_wguide.Eig_values[0]*((wl_nm*1e-9)/(2.*np.pi)))
print "n_eff = ", np.round(n_eff_sim, 4)

# Choose acoustic wavenumber to solve for
# Backward SBS
# AC mode couples EM modes on +ve to -ve lightline, hence factor 2.
k_AC = 2*np.real(sim_EM_wguide.Eig_values[0])
print 'AC wavenumber (1/m) = ', np.round(k_AC, 4)
# Forward (intramode) SBS
# EM modes on same lightline.
# k_AC = 0.0

# Calculate Acoustic Modes
sim_AC_wguide = wguide.calc_AC_modes(wl_nm, num_AC_modes,
        k_AC=k_AC, EM_sim=sim_EM_wguide)
# Print the frequencies of AC modes.
print 'Freq of AC modes (GHz) \n', np.round(np.real(sim_AC_wguide.Eig_values)*1e-9, 4)
# Plot the AC modes fields, important to specify this with EM_AC='AC'.
# The AC modes are calculated on a subset of the full unitcell,
# which excludes vacuum regions, so no need to restrict area plotted.
plotting.plt_mode_fields(sim_AC_wguide, EM_AC='AC')

# Calculate interaction integrals and SBS gain for PE and MB effects combined,
# as well as just for PE, and just for MB. Also calculate acoustic loss alpha.
SBS_gain, SBS_gain_PE, SBS_gain_MB, alpha = integration.gain_and_qs(
    sim_EM_wguide, sim_AC_wguide, k_AC,
    EM_ival1=EM_ival1, EM_ival2=EM_ival2, AC_ival=AC_ival)
# Print the Backward SBS gain of the AC modes.
print "SBS_gain PE contribution \n", SBS_gain_PE[EM_ival1,EM_ival2,:]/alpha
print "SBS_gain MB contribution \n", SBS_gain_MB[EM_ival1,EM_ival2,:]/alpha
```

```
print "SBS_gain total \n", SBS_gain[EM_ival1,EM_ival2,:]/alpha
# Mask negligible gain values to improve clarity of print out.
threshold = 1e-3
masked_PE = np.ma.masked_inside(SBS_gain_PE[EM_ival1,EM_ival2,:]/alpha, 0, threshold)
masked_MB = np.ma.masked_inside(SBS_gain_MB[EM_ival1,EM_ival2,:]/alpha, 0, threshold)
masked = np.ma.masked_inside(SBS_gain[EM_ival1,EM_ival2,:]/alpha, 0, threshold)
print "\n"
print "SBS_gain PE contribution \n", masked_PE
print "SBS_gain MB contribution \n", masked_MB
print "SBS_gain total \n", masked

end = time.time()
print(end - start)
```

SBS Gain Spectra

```
plt.close()
""" Calculate the backward SBS gain spectra of a
    silicon waveguide surrounded in air.

    Show how to save simulation objects (eg EM mode calcs)
    to expedite the process of altering later parts of
    simulations.
"""

import time
import datetime
import numpy as np
import sys
sys.path.append("../backend/")
import matplotlib
matplotlib.use('pdf')
import matplotlib.pyplot as plt

import materials
import objects
import mode_calcs
import integration
import plotting
from fortran import NumBAT


# Geometric Parameters - all in nm.
wl_nm = 1550
unitcell_x = 2.5*wl_nm
unitcell_y = unitcell_x
inc_a_x = 314.7
inc_a_y = 0.9*inc_a_x
inc_shape = 'rectangular'

# Optical Parameters
eps = 12.25
num_EM_modes = 20
num_AC_modes = 20
EM_ival1=0
EM_ival2=EM_ival1
```

```
AC_ival='All'

# Acoustic Parameters
s = 2330  # kg/m3
c_11 = 165.7e9; c_12 = 63.9e9; c_44 = 79.6e9  # Pa
p_11 = -0.094; p_12 = 0.017; p_44 = -0.051
eta_11 = 5.9e-3 ; eta_12 = 5.16e-3 ; eta_44 = 0.620e-3  # Pa
inc_a_AC_props = [s, c_11, c_12, c_44, p_11, p_12, p_44,
                  eta_11, eta_12, eta_44]

# Use all specified parameters to create a waveguide object.
wguide = objects.Struct(unitcell_x,inc_a_x,unitcell_y,inc_a_y,inc_shape,
                        bkg_material=materials.Material(1.0 + 0.0j),
                        inc_a_material=materials.Material(np.sqrt(eps)),
                        loss=False, inc_a_AC=inc_a_AC_props,
                        lc_bkg=2, lc2=2000.0, lc3=10.0)


# Expected effective index of fundamental guided mode.
n_eff = np.real(np.sqrt(eps))-0.1

# Calculate Electromagnetic Modes
sim_EM_wguide = wguide.calc_EM_modes(wl_nm, num_EM_modes, n_eff)
# Save calculated :Simmo: object for EM calculation.
np.savez('wguide_data', sim_EM_wguide=sim_EM_wguide)

# The previous two lines can be commented out and the following
# two uncommented to provide precisely the same objects for the
# remainder of the simulation.
# npzfile = np.load('wguide_data.npz')
# sim_EM_wguide = npzfile['sim_EM_wguide'].tolist()

# Print the wavevectors of EM modes.
print 'k_z of EM modes \n', np.round(np.real(sim_EM_wguide.Eig_values), 4)

# Calculate the EM effective index of the waveguide.
n_eff_sim = np.real(sim_EM_wguide.Eig_values[0]*((wl_nm*1e-9)/(2.*np.pi)))
print "n_eff", np.round(n_eff_sim, 4)

# Choose acoustic wavenumber to solve for
# Backward SBS
# AC mode couples EM modes on +ve to -ve lightline, hence factor 2.
k_AC = 2*np.real(sim_EM_wguide.Eig_values[0])

# Calculate Acoustic Modes
sim_AC_wguide = wguide.calc_AC_modes(wl_nm, num_AC_modes, k_AC,
    EM_sim=sim_EM_wguide)
# np.savez('wguide_data_AC', sim_AC_wguide=sim_AC_wguide)

# The previous two lines can be commented out and the following
# two uncommented to provide precisely the same objects for the
# remainder of the simulation.
# npzfile = np.load('wguide_data_AC.npz')
# sim_AC_wguide = npzfile['sim_AC_wguide'].tolist()

# Print the frequencies of AC modes.
print 'Freq of AC modes (GHz) \n', np.round(np.real(sim_AC_wguide.Eig_values)*1e-9, 4)
```

```
# Do not calculate the acoustic loss from our fields, but instead set a
# predetirmined Q factor. (Useful for instance when replicating others results).
set_q_factor = 1000.

# Calculate interaction integrals and SBS gain for PE and MB effects combined,
# as well as just for PE, and just for MB. Also calculate acoustic loss alpha.
SBS_gain, SBS_gain_PE, SBS_gain_MB, alpha = integration.gain_and_qs(
    sim_EM_wguide, sim_AC_wguide, k_AC,
    EM_ival1=EM_ival1, EM_ival2=EM_ival2, AC_ival=AC_ival, fixed_Q=set_q_factor)
np.savez('wguide_data_AC_gain', SBS_gain=SBS_gain, alpha=alpha)

# The previous two lines can be commented out and the following
# three uncommented to provide precisely the same objects for the
# remainder of the simulation.
# npzfile = np.load('wguide_data_AC_gain.npz')
# SBS_gain = npzfile['SBS_gain']
# alpha = npzfile['alpha']


# Construct the SBS gain spectrum, built up
# from Lorentzian peaks of the individual modes.
tune_steps = 5e4
tune_range = 10 # GHz
# Construct an odd range of frequencies that is guaranteed to include
# the central resonance frequency.
detuning_range = np.append(np.linspace(-1*tune_range, 0, tune_steps),
                   np.linspace(0, tune_range, tune_steps)[1:])*1e9 # GHz
# Line width of resonances should be v_g * alpha,
# but we don't have convenient access to v_g, therefore
# phase velocity as approximation to group velocity
phase_v = sim_AC_wguide.Eig_values/k_AC
linewidth = phase_v*alpha

freq_min = 10  # GHz
freq_max = 25  # GHz
interp_grid_points = 10000
interp_grid = np.linspace(freq_min, freq_max, interp_grid_points)
interp_values = np.zeros(interp_grid_points)

plt.figure()
plt.clf()
for AC_i in range(len(alpha)):
    gain_list = np.real(SBS_gain[EM_ival1,EM_ival2,AC_i]/alpha[AC_i]
                  *linewidth[AC_i]**2/(linewidth[AC_i]**2 + detuning_range**2))
    freq_list_GHz = np.real(sim_AC_wguide.Eig_values[AC_i] + detuning_range)*1e-9
    plt.plot(freq_list_GHz, gain_list)
    # set up an interpolation for summing all the gain peaks
    interp_spectrum = np.interp(interp_grid, freq_list_GHz, gain_list)
    interp_values += interp_spectrum
plt.plot(interp_grid, interp_values, 'k', linewidth=3, label="Total")
plt.legend(loc=0)
plt.xlim(freq_min,freq_max)
plt.xlabel('Frequency (GHz)')
plt.ylabel('Gain (1/Wm)')
plt.savefig('gain_spectra-mode_comps.pdf')
plt.close()

freq_min_zoom = 12
```

```
freq_max_zoom = 14
interp_values = np.zeros(interp_grid_points)
interp_values_PE = np.zeros(interp_grid_points)
interp_values_MB = np.zeros(interp_grid_points)
plt.figure()
plt.clf()
for AC_i in range(len(alpha)):
    gain_list = np.real(SBS_gain[EM_ival1,EM_ival2,AC_i]/alpha[AC_i]
                *linewidth[AC_i]**2/(linewidth[AC_i]**2 + detuning_range**2))
    freq_list_GHz = np.real(sim_AC_wguide.Eig_values[AC_i] + detuning_range)*1e-9
    interp_spectrum = np.interp(interp_grid, freq_list_GHz, gain_list)
    interp_values += interp_spectrum

    gain_list_PE = np.real(SBS_gain_PE[EM_ival1,EM_ival2,AC_i]/alpha[AC_i]
                *linewidth[AC_i]**2/(linewidth[AC_i]**2 + detuning_range**2))
    interp_spectrum_PE = np.interp(interp_grid, freq_list_GHz, gain_list_PE)
    interp_values_PE += interp_spectrum_PE

    gain_list_MB = np.real(SBS_gain_MB[EM_ival1,EM_ival2,AC_i]/alpha[AC_i]
                *linewidth[AC_i]**2/(linewidth[AC_i]**2 + detuning_range**2))
    interp_spectrum_MB = np.interp(interp_grid, freq_list_GHz, gain_list_MB)
    interp_values_MB += interp_spectrum_MB
plt.plot(interp_grid, interp_values, 'k', linewidth=3, label="Total")
plt.plot(interp_grid, interp_values_PE, 'r', linewidth=3, label="PE")
plt.plot(interp_grid, interp_values_MB, 'g', linewidth=3, label="MB")
plt.legend(loc=0)
plt.xlim(freq_min_zoom,freq_max_zoom)
plt.xlabel('Frequency (GHz)')
plt.ylabel('Gain (1/Wm)')
plt.savefig('gain_spectra-MB_PE_comps.pdf')
plt.close()
```

Investigating Dispersion

```
plt.close()
""" Calculate a dispersion diagram of the acoustic modes
    from k_AC = 0 (forward SBS) to k_AC = 2*k_EM (backward SBS).
"""

import time
import datetime
import numpy as np
import sys
sys.path.append("../backend/")
import matplotlib
matplotlib.use('pdf')
import matplotlib.pyplot as plt

import materials
import objects
import mode_calcs
import integration
import plotting
from fortran import NumBAT
```

```
# Geometric Parameters - all in nm.
wl_nm = 1550
unitcell_x = 2.5*wl_nm
unitcell_y = unitcell_x
inc_a_x = 314.7
inc_a_y = 0.9*inc_a_x
inc_shape = 'rectangular'

# Optical Parameters
eps = 12.25
num_EM_modes = 20
num_AC_modes = 20
EM_ival1=0
EM_ival2=EM_ival1
AC_ival='All'

# Acoustic Parameters
s = 2330  # kg/m3
c_11 = 165.7e9; c_12 = 63.9e9; c_44 = 79.6e9  # Pa
p_11 = -0.094; p_12 = 0.017; p_44 = -0.051
eta_11 = 5.9e-3 ; eta_12 = 5.16e-3 ; eta_44 = 0.620e-3  # Pa
inc_a_AC_props = [s, c_11, c_12, c_44, p_11, p_12, p_44,
                  eta_11, eta_12, eta_44]

# Use all specified parameters to create a waveguide object.
wguide = objects.Struct(unitcell_x,inc_a_x,unitcell_y,inc_a_y,inc_shape,
                        bkg_material=materials.Material(1.0 + 0.0j),
                        inc_a_material=materials.Material(np.sqrt(eps)),
                        loss=False, inc_a_AC=inc_a_AC_props,
                        lc_bkg=2, lc2=2000.0, lc3=10.0)


# Expected effective index of fundamental guided mode.
n_eff = np.real(np.sqrt(eps))-0.1

# Calculate Electromagnetic Modes
# sim_EM_wguide = wguide.calc_EM_modes(wl_nm, num_EM_modes, n_eff)
# np.savez('wguide_data', sim_EM_wguide=sim_EM_wguide)

# Assuming this calculation is run directly after simo-tut_02
# we don't need to recalculate EM modes, but can load them in.
npzfile = np.load('wguide_data.npz')
sim_EM_wguide = npzfile['sim_EM_wguide'].tolist()


# Will scan from forward to backward SBS,
# so need to know k_AC of backward SBS.
k_AC = 2*sim_EM_wguide.Eig_values[0]
# Number of wavevectors steps.
nu_ks = 20

plt.clf()
plt.figure(figsize=(10,6))
ax = plt.subplot(1,1,1)
for q_ac in np.linspace(0.0,k_AC,nu_ks):
    sim_AC_wguide = wguide.calc_AC_modes(wl_nm, num_AC_modes, q_ac, EM_sim=sim_EM_wguide)
    prop_AC_modes = np.array([np.real(x) for x in sim_AC_wguide.Eig_values if abs(np.real(x)) > abs(n
    sym_list = integration.symmetries(sim_AC_wguide)
```

```
    for i in range(len(prop_AC_modes)):
        Om = prop_AC_modes[i]*1e-9
        if sym_list[i][0] == 1 and sym_list[i][1] == 1 and sym_list[i][2] == 1:
            sym_A, = plt.plot(np.real(q_ac/k_AC), Om, 'or')
        if sym_list[i][0] == -1 and sym_list[i][1] == 1 and sym_list[i][2] == -1:
            sym_B, = plt.plot(np.real(q_ac/k_AC), Om, 'vc')
        if sym_list[i][0] == 1 and sym_list[i][1] == -1 and sym_list[i][2] == -1:
            sym_C, = plt.plot(np.real(q_ac/k_AC), Om, 'sb')
        if sym_list[i][0] == -1 and sym_list[i][1] == -1 and sym_list[i][2] == 1:
            sym_D, = plt.plot(np.real(q_ac/k_AC), Om, '^g')
ax.set_ylim(0,20)
ax.set_xlim(0,1)
plt.legend([sym_A, sym_B, sym_C, sym_D],['E',r'C$_2$',r'$\sigma_y$',r'$\sigma_x$'], loc='lower right
plt.xlabel(r'Axial wavevector (normalised)')
plt.ylabel(r'Frequency (GHz)')
plt.savefig('symetrised_dispersion.pdf', bbox_inches='tight')
plt.close()
```

Parameter Scan of Widths

```
plt.close()
""" Calculate the backward SBS gain spectra as a function of
    waveguide width, for silicon waveguides surrounded in air.

    Also shows how to use python multiprocessing library.
"""

import time
import datetime
import numpy as np
import sys
from multiprocessing import Pool
sys.path.append("../backend/")
import matplotlib
matplotlib.use('pdf')
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.collections import PolyCollection
from matplotlib.colors import colorConverter

import materials
import objects
import mode_calcs
import integration
import plotting
from fortran import NumBAT


# Select the number of CPUs to use in simulation.
num_cores = 6

# Geometric Parameters - all in nm.
wl_nm = 1550
inc_shape = 'rectangular'
```

```
# Optical Parameters
eps = 12.25
num_EM_modes = 20
num_AC_modes = 20
EM_ival1=0
EM_ival2=EM_ival1
AC_ival='All'

# Acoustic Parameters
s = 2330  # kg/m3
c_11 = 165.7e9; c_12 = 63.9e9; c_44 = 79.6e9  # Pa
p_11 = -0.094; p_12 = 0.017; p_44 = -0.051
eta_11 = 5.9e-3 ; eta_12 = 5.16e-3 ; eta_44 = 0.620e-3  # Pa
inc_a_AC_props = [s, c_11, c_12, c_44, p_11, p_12, p_44,
                  eta_11, eta_12, eta_44]

# Width previous simo's done for, with known meshing params
known_geo = 315.

def modes_n_gain(wguide):
    # Expected effective index of fundamental guided mode.
    n_eff = (np.real(np.sqrt(eps))-0.1) * wguide.inc_a_x/known_geo
    # Calculate Electromagnetic Modes
    sim_EM_wguide = wguide.calc_EM_modes(wl_nm, num_EM_modes, n_eff)
    # Backward SBS
    k_AC = 2*np.real(sim_EM_wguide.Eig_values[0])
    # Calculate Acoustic Modes
    sim_AC_wguide = wguide.calc_AC_modes(wl_nm, num_AC_modes, k_AC,
        EM_sim=sim_EM_wguide)
    # Calculate interaction integrals and SBS gain
    SBS_gain, SBS_gain_PE, SBS_gain_MB, alpha = integration.gain_and_qs(
        sim_EM_wguide, sim_AC_wguide, k_AC,
        EM_ival1=EM_ival1, EM_ival2=EM_ival2, AC_ival=AC_ival)

    return [sim_EM_wguide, sim_AC_wguide, SBS_gain, SBS_gain_PE, SBS_gain_MB, alpha, k_AC]


nu_widths = 12
waveguide_widths = np.linspace(300,400,nu_widths)
geo_objects_list = []
# Scale meshing to new structures
for width in waveguide_widths:
    msh_ratio = (width/known_geo)
    # Geometric Parameters - all in nm.
    unitcell_x = 2.5*wl_nm*msh_ratio
    unitcell_y = unitcell_x
    inc_a_x = width
    inc_a_y = 0.9*inc_a_x

    # Use all specified parameters to create a waveguide object.
    wguide = objects.Struct(unitcell_x,inc_a_x,unitcell_y,
                            inc_a_y,inc_shape,
                            bkg_material=materials.Material(1.0 + 0.0j),
                            inc_a_material=materials.Material(np.sqrt(eps)),
                            loss=False, inc_a_AC=inc_a_AC_props,
                            lc_bkg=2, lc2=2000.0, lc3=10.0)
    geo_objects_list.append(wguide)
```

```
# Run widths in parallel across num_cores CPUs using multiprocessing package.
pool = Pool(num_cores)
width_objs = pool.map(modes_n_gain, geo_objects_list)
# np.savez('Simo_results', width_objs=width_objs)
# npzfile = np.load('Simo_results.npz')
# width_objs = npzfile['width_objs'].tolist()


n_effs = []
freqs_gains = []
interp_grid_points = 10000
interp_grid = np.linspace(10, 25, interp_grid_points)
for i_w, width_obj in enumerate(width_objs):
    interp_values = np.zeros(interp_grid_points)
    sim_EM = width_obj[0]
    sim_AC = width_obj[1]
    SBS_gain = width_obj[2]
    alpha = width_obj[5]
    k_AC = width_obj[6]
    # Calculate the EM effective index of the waveguide (k_AC = 2*k_EM).
    n_eff_sim = round(np.real((k_AC/2.)*((wl_nm*1e-9)/(2.*np.pi))), 4)
    n_effs.append(n_eff_sim)

    # Construct the SBS gain spectrum, built up
    # from Lorentzian peaks of the individual modes.
    tune_steps = 5e4
    tune_range = 10 # GHz
    # Construct an odd range of frequencies that is guaranteed to include
    # the central resonance frequency
    detuning_range = np.append(np.linspace(-1*tune_range, 0, tune_steps),
                        np.linspace(0, tune_range, tune_steps)[1:])*1e9 # GHz
    phase_v = sim_AC.Eig_values/k_AC
    line_width = phase_v*alpha

    plt.figure(figsize=(13,13))
    plt.clf()
    for AC_i in range(len(alpha)):
        gain_list = np.real(SBS_gain[EM_ival1,EM_ival2,AC_i]/alpha[AC_i]
                    *line_width[AC_i]**2/(line_width[AC_i]**2 + detuning_range**2))
        freq_list_GHz = np.real(sim_AC.Eig_values[AC_i] + detuning_range)*1e-9
        plt.plot(freq_list_GHz, gain_list,linewidth=3)
        # set up an interpolation for summing all the gain peaks
        interp_spectrum = np.interp(interp_grid, freq_list_GHz, gain_list)
        interp_values += interp_spectrum
    freqs_gains.append(zip(interp_grid, interp_values))
    plt.plot(interp_grid, interp_values, 'k', linewidth=4)
    plt.xlim(10,25)
    plt.xlabel('Frequency (GHz)')
    plt.ylabel('Gain (1/Wm)')
    plt.savefig('gain_spectra_%i.pdf' % i_w)
    plt.close()

print 'Widths', waveguide_widths
print 'n_effs', n_effs

# Plot a 'waterfall' plot.
fig = plt.figure()
ax = fig.gca(projection='3d')
```

```
poly = PolyCollection(freqs_gains)
poly.set_alpha(0.7)
ax.add_collection3d(poly, zs=waveguide_widths, zdir='y')
ax.set_xlabel('Frequency (GHz)', fontsize=14)
ax.set_xlim3d(10,25)
ax.set_ylabel('Width (nm)', fontsize=14)
ax.set_ylim3d(waveguide_widths[0], waveguide_widths[-1])
ax.set_zlabel('Gain (1/Wm)', fontsize=14)
ax.set_zlim3d(0,1500)
# We change the fontsize of minor ticks label
plt.tick_params(axis='both', which='major', labelsize=12, pad=-2)
plt.savefig('gain_spectra_waterfall.pdf')
plt.close()
```

Convergence Study

```
print "Calculation time", time_list
""" Calculate the convergence as a function of FEM mesh
    for backward SBS gain spectra of a
    silicon waveguide surrounded in air.

    Again utilises python multiprocessing.
"""

import time
import datetime
import numpy as np
import sys
sys.path.append("../backend/")
import matplotlib
matplotlib.use('pdf')
import matplotlib.pyplot as plt

import materials
import objects
import mode_calcs
import integration
import plotting
from fortran import NumBAT


# Select the number of CPUs to use in simulation.
num_cores = 6

# Geometric Parameters - all in nm.
wl_nm = 1550
unitcell_x = 2.5*wl_nm
unitcell_y = unitcell_x
inc_a_x = 314.7
inc_a_y = 0.9*inc_a_x
inc_shape = 'rectangular'

# Optical Parameters
eps = 12.25
num_EM_modes = 20
num_AC_modes = 20
```

```
EM_ival1=0
EM_ival2=EM_ival1
AC_ival='All'

# Acoustic Parameters
s = 2330  # kg/m3
c_11 = 165.7e9; c_12 = 63.9e9; c_44 = 79.6e9  # Pa
p_11 = -0.094; p_12 = 0.017; p_44 = -0.051
eta_11 = 5.9e-3 ; eta_12 = 5.16e-3 ; eta_44 = 0.620e-3  # Pa
inc_a_AC_props = [s, c_11, c_12, c_44, p_11, p_12, p_44,
                  eta_11, eta_12, eta_44]


nu_lcs = 6
lc_bkg_list = 2*np.ones(nu_lcs)
lc_list = np.linspace(2e2,6e3,nu_lcs)
x_axis = lc_bkg_list
x_axis = lc_list
conv_list = []
time_list = []
# Do not run in parallel, otherwise there are confusions reading the msh files!
for i_lc, lc_ref in enumerate(lc_list):
    start = time.time()
    print i_lc+1, "/", nu_lcs
    lc3 = 0.01*lc_ref
    lc_bkg = lc_bkg_list[i_lc]
    wguide = objects.Struct(unitcell_x,inc_a_x,unitcell_y,
                            inc_a_y,inc_shape,
                            bkg_material=materials.Material(1.0 + 0.0j),
                            inc_a_material=materials.Material(np.sqrt(eps)),
                            loss=False, inc_a_AC=inc_a_AC_props,
                            lc_bkg=lc_bkg, lc2=lc_ref, lc3=lc3, force_mesh=True)

    # Expected effective index of fundamental guided mode.
    n_eff = np.real(np.sqrt(eps))-0.1
    # Calculate Electromagnetic Modes
    sim_EM_wguide = wguide.calc_EM_modes(wl_nm, num_EM_modes, n_eff)
    # Backward SBS
    k_AC = 2*np.real(sim_EM_wguide.Eig_values[0])
    # Calculate Acoustic Modes
    sim_AC_wguide = wguide.calc_AC_modes(wl_nm, num_AC_modes, k_AC,
        EM_sim=sim_EM_wguide)
    # Calculate interaction integrals and SBS gain
    SBS_gain, SBS_gain_PE, SBS_gain_MB, alpha = integration.gain_and_qs(
        sim_EM_wguide, sim_AC_wguide, k_AC,
        EM_ival1=EM_ival1, EM_ival2=EM_ival2, AC_ival=AC_ival)

    conv_list.append([sim_EM_wguide, sim_AC_wguide, SBS_gain, SBS_gain_PE, SBS_gain_MB, alpha])
    end = time.time()
    time_list.append(end - start)


# np.savez('Simo_results', conv_list=conv_list)
# npzfile = np.load('Simo_results.npz')
# conv_list = npzfile['conv_list'].tolist()

# sim_EM_wguide = conv_list[-1][0]
# # print 'k_z of EM wave \n', sim_EM_wguide.Eig_values
```

```
# plotting.plt_mode_fields(sim_EM_wguide, xlim=0.4, ylim=0.4, EM_AC='EM')

rel_modes = [2,4,8]
rel_mode_freq_EM = np.zeros(nu_lcs,dtype=complex)
rel_mode_freq_AC = np.zeros((nu_lcs,len(rel_modes)),dtype=complex)
rel_mode_gain = np.zeros((nu_lcs,len(rel_modes)),dtype=complex)
rel_mode_gain_MB = np.zeros((nu_lcs,len(rel_modes)),dtype=complex)
rel_mode_gain_PE = np.zeros((nu_lcs,len(rel_modes)),dtype=complex)
# rel_mode_alpha = np.zeros((nu_lcs,len(rel_modes)),dtype=complex)
for i_conv, conv_obj in enumerate(conv_list):
    rel_mode_freq_EM[i_conv] = conv_obj[0].Eig_values[0]
    for i_m, rel_mode in enumerate(rel_modes):
        rel_mode_freq_AC[i_conv,i_m] = conv_obj[1].Eig_values[rel_mode]
        rel_mode_gain[i_conv,i_m] = conv_obj[2][EM_ival1,EM_ival2,rel_mode]/conv_obj[5][rel_mode]
        rel_mode_gain_PE[i_conv,i_m] = conv_obj[3][EM_ival1,EM_ival2,rel_mode]/conv_obj[5][rel_mode]
        rel_mode_gain_MB[i_conv,i_m] = conv_obj[4][EM_ival1,EM_ival2,rel_mode]/conv_obj[5][rel_mode]



xlabel = "Mesh Refinement Factor"
fig = plt.figure()
plt.clf()
ax1 = fig.add_subplot(1,1,1)
ax2 = ax1.twinx()
ax2.yaxis.tick_left()
ax2.yaxis.set_label_position("left")
EM_plot_Mk = rel_mode_freq_EM*1e-6
error0 = np.abs((np.array(EM_plot_Mk[0:-1])-EM_plot_Mk[-1])/EM_plot_Mk[-1])
ax2.plot(x_axis[0:-1], error0, 'b-v',label='Mode #%i'%EM_ival1)
ax1.plot(x_axis, np.real(EM_plot_Mk), 'r-.o',label=r'EM k$_z$')
ax1.yaxis.tick_right()
ax1.spines['right'].set_color('red')
ax1.yaxis.label.set_color('red')
ax1.yaxis.set_label_position("right")
ax1.tick_params(axis='y', colors='red')
handles, labels = ax2.get_legend_handles_labels()
ax2.legend(handles, labels)
ax1.set_xlabel(xlabel)
ax1.set_ylabel(r"EM k$_z$ ($\times 10^6$ 1/m)")
ax2.set_ylabel(r"Relative Error EM k$_z$")
ax2.set_yscale('log', nonposx='clip')
plt.savefig('convergence-freq_EM.pdf', bbox_inches='tight')
plt.close()

fig = plt.figure()
plt.clf()
ax1 = fig.add_subplot(1,1,1)
ax2 = ax1.twinx()
ax2.yaxis.tick_left()
ax2.yaxis.set_label_position("left")
for i_m, rel_mode in enumerate(rel_modes):
    rel_mode_freq_AC_plot_GHz = rel_mode_freq_AC[:,i_m]*1e-9
    error0 = np.abs((np.array(rel_mode_freq_AC_plot_GHz[0:-1])-rel_mode_freq_AC_plot_GHz[-1])/rel_mod
    ax2.plot(x_axis[0:-1], error0, '-v',label='Mode #%i'%rel_mode)
    ax1.plot(x_axis, np.real(rel_mode_freq_AC_plot_GHz), '-.o',label=r'AC Freq mode #%i'%rel_mode)
ax1.yaxis.tick_right()
ax1.spines['right'].set_color('red')
ax1.yaxis.label.set_color('red')
```

```
ax1.yaxis.set_label_position("right")
ax1.tick_params(axis='y', colors='red')
handles, labels = ax2.get_legend_handles_labels()
ax2.legend(handles, labels)
ax1.set_xlabel(xlabel)
ax1.set_ylabel(r"AC Freq (GHz)")
ax2.set_ylabel(r"Relative Error AC Freq")
ax2.set_yscale('log', nonposx='clip')
plt.savefig('convergence-freq_AC.pdf', bbox_inches='tight')
plt.close()

fig = plt.figure()
plt.clf()
ax1 = fig.add_subplot(1,1,1)
ax2 = ax1.twinx()
ax2.yaxis.tick_left()
ax2.yaxis.set_label_position("left")
for i_m, rel_mode in enumerate(rel_modes):
    rel_mode_gain_plot = rel_mode_gain[:,i_m]
    error0 = np.abs((np.array(rel_mode_gain_plot[0:-1])-rel_mode_gain_plot[-1])/rel_mode_gain_plot[-1
    ax2.plot(x_axis[0:-1], error0, '-v',label=r'Mode #%i'%rel_mode)
    ax1.plot(x_axis, np.real(rel_mode_gain_plot), '-.o',label=r'Gain mode #%i'%rel_mode)
ax1.yaxis.tick_right()
ax1.spines['right'].set_color('red')
ax1.yaxis.label.set_color('red')
ax1.yaxis.set_label_position("right")
ax1.tick_params(axis='y', colors='red')
handles, labels = ax2.get_legend_handles_labels()
ax2.legend(handles, labels)
ax1.set_xlabel(xlabel)
ax1.set_ylabel(r"Gain")
ax2.set_ylabel(r"Relative Error Gain")
ax2.set_yscale('log', nonposx='clip')
plt.savefig('convergence-Gain.pdf', bbox_inches='tight')
plt.close()

fig = plt.figure()
plt.clf()
ax1 = fig.add_subplot(1,1,1)
ax2 = ax1.twinx()
ax2.yaxis.tick_left()
ax2.yaxis.set_label_position("left")
for i_m, rel_mode in enumerate(rel_modes):
    rel_mode_gain_PE_plot = rel_mode_gain_PE[:,i_m]
    error0 = np.abs((np.array(rel_mode_gain_PE_plot[0:-1])-rel_mode_gain_PE_plot[-1])/rel_mode_gain_P
    ax2.plot(x_axis[0:-1], error0, '-v',label=r'Mode #%i'%rel_mode)
    ax1.plot(x_axis, np.real(rel_mode_gain_PE_plot), '-.o',label=r'Gain mode #%i'%rel_mode)
ax1.yaxis.tick_right()
ax1.spines['right'].set_color('red')
ax1.yaxis.label.set_color('red')
ax1.yaxis.set_label_position("right")
ax1.tick_params(axis='y', colors='red')
handles, labels = ax2.get_legend_handles_labels()
ax2.legend(handles, labels)
ax1.set_xlabel(xlabel)
ax1.set_ylabel(r"Gain")
ax2.set_ylabel(r"Relative Error Gain")
ax2.set_yscale('log', nonposx='clip')
```

```
plt.savefig('convergence-Gain_PE.pdf', bbox_inches='tight')
plt.close()

fig = plt.figure()
plt.clf()
ax1 = fig.add_subplot(1,1,1)
ax2 = ax1.twinx()
ax2.yaxis.tick_left()
ax2.yaxis.set_label_position("left")
for i_m, rel_mode in enumerate(rel_modes):
    rel_mode_gain_MB_plot = rel_mode_gain_MB[:,i_m]
    error0 = np.abs((np.array(rel_mode_gain_MB_plot[0:-1])-rel_mode_gain_MB_plot[-1])/rel_mode_gain_M
    ax2.plot(x_axis[0:-1], error0, '-v',label=r'Mode #%i'%rel_mode)
    ax1.plot(x_axis, np.real(rel_mode_gain_MB_plot), '-.o',label=r'Gain mode #%i'%rel_mode)
ax1.yaxis.tick_right()
ax1.spines['right'].set_color('red')
ax1.yaxis.label.set_color('red')
ax1.yaxis.set_label_position("right")
ax1.tick_params(axis='y', colors='red')
handles, labels = ax2.get_legend_handles_labels()
ax2.legend(handles, labels)
ax1.set_xlabel(xlabel)
ax1.set_ylabel(r"Gain")
ax2.set_ylabel(r"Relative Error Gain")
ax2.set_yscale('log', nonposx='clip')
plt.savefig('convergence-Gain_MB.pdf', bbox_inches='tight')
plt.close()

print "Calculation time", time_list
```

Embedded Chalcogenide Example

```
plt.close()
""" Calculate the backward SBS gain spectra of a
    chalcogenide (As2S3) waveguide surrounded in silica.
"""

import time
import datetime
import numpy as np
import sys
sys.path.append("../backend/")
import matplotlib
matplotlib.use('pdf')
import matplotlib.pyplot as plt

import materials
import objects
import mode_calcs
import integration
import plotting
from fortran import NumBAT


# Geometric Parameters - all in nm.
wl_nm = 1550
```

```
unitcell_x = 6.5*wl_nm
unitcell_y = unitcell_x
inc_a_x = 1900
inc_a_y = 680
inc_shape = 'rectangular'

# Optical parameters
n_b = 1.44
n_i = 2.44
num_EM_modes = 20
# There are lots of leaky acoustic modes!
num_AC_modes = 350
EM_ival1=0
EM_ival2=EM_ival1
AC_ival='All'

# Acoustic Parameters
# Background - silca
# # Use the isotropic parameters for silca.
# s = 2200  # kg/m3
# E = 7.3e10
# v = 0.17
# c_11, c_12, c_44 = materials.isotropic_stiffness(E, v)
# p_11 = 0.121; p_12 = 0.270; p_44 = -0.075

# Silca - Laude AIP Advances 2013
s = 2203  # kg/m3
c_11 = 78e9; c_12 = 16e9; c_44 = 31e9
p_11 = 0.12; p_12 = 0.270; p_44 = -0.073

eta_11 = 1.6e-3 ; eta_12 = 1.29e-3 ; eta_44 = 0.16e-3  # Pa s
bkg_AC_props = [s, c_11, c_12, c_44, p_11, p_12, p_44,
                  eta_11, eta_12, eta_44]
# Inclusion a - As2S3
s = 3210  # kg/m3
c_11 = 2.104e10; c_12 = 8.363e9; c_44 =6.337e9 # Pa
p_11 = 0.25; p_12 = 0.24; p_44 = 0.005
eta_11 = 9e-3 ; eta_12 = 7.5e-3 ; eta_44 = 0.75e-3  # Pa s
inc_a_AC_props = [s, c_11, c_12, c_44, p_11, p_12, p_44,
                  eta_11, eta_12, eta_44]

# Use all specified parameters to create a waveguide object.
wguide = objects.Struct(unitcell_x,inc_a_x,unitcell_y,inc_a_y,inc_shape,
                        bkg_material=materials.Material(n_b),
                        inc_a_material=materials.Material(n_i),
                        loss=False,
                        bkg_AC=bkg_AC_props,
                        inc_a_AC=inc_a_AC_props,plotting_fields=False,
                        lc_bkg=3, lc2=2500.0, lc3=10.0)

# Expected effective index of fundamental guided mode.
n_eff = 2.

# Calculate Electromagnetic Modes
# sim_EM_wguide = wguide.calc_EM_modes(wl_nm, num_EM_modes, n_eff=n_eff)
# np.savez('wguide_data-chalc', sim_EM_wguide=sim_EM_wguide)
npzfile = np.load('wguide_data-chalc.npz')
sim_EM_wguide = npzfile['sim_EM_wguide'].tolist()
```

```
# plotting.plt_mode_fields(sim_EM_wguide, xlim=0.4, ylim=0.4, EM_AC='EM', add_name='As2S3')

# Print the wavevectors of EM modes.
print 'k_z of EM wave \n', np.round(np.real(sim_EM_wguide.Eig_values), 4)
n_eff_sim = np.real((sim_EM_wguide.Eig_values[0]*((wl_nm*1e-9)/(2.*np.pi))))
print 'n_eff of fund. EM mode \n', np.round(n_eff_sim, 4)
n_eff_sim = np.real((sim_EM_wguide.Eig_values[2]*((wl_nm*1e-9)/(2.*np.pi))))
print 'n_eff of 3rd EM mode \n', np.round(n_eff_sim, 4)

# Choose acoustic wavenumber to solve for
# Backward SBS
# AC mode couples EM modes on +ve to -ve lightline, hence factor 2.
k_AC = 2*np.real(sim_EM_wguide.Eig_values[0])

# Calculate Acoustic Modes
# sim_AC_wguide = wguide.calc_AC_modes(wl_nm, num_AC_modes, k_AC=k_AC,
#     EM_sim=sim_EM_wguide)
# np.savez('wguide_data_AC', sim_AC_wguide=sim_AC_wguide)
npzfile = np.load('wguide_data_AC.npz')
sim_AC_wguide = npzfile['sim_AC_wguide'].tolist()
# plotting.plt_mode_fields(sim_AC_wguide, xlim=0.3, ylim=0.3, EM_AC='AC', add_name='As2S3')

# Print the frequencies of AC modes.
print 'Res freq of AC wave (GHz) \n', np.round(np.real(sim_AC_wguide.Eig_values*1e-9), 4)

# Experimentaly obtained Q factor.
set_q_factor = 1600.

SBS_gain, SBS_gain_PE, SBS_gain_MB, alpha = integration.gain_and_qs(
    sim_EM_wguide, sim_AC_wguide, k_AC,
    EM_ival1=EM_ival1, EM_ival2=EM_ival2, AC_ival=AC_ival, fixed_Q=set_q_factor)
# np.savez('wguide_data_AC_gain', SBS_gain=SBS_gain, alpha=alpha)
# npzfile = np.load('wguide_data_AC_gain.npz')
# SBS_gain = npzfile['SBS_gain']
# alpha = npzfile['alpha']


# Construct the SBS gain spectrum, built up
# from Lorentzian peaks of the individual modes.
tune_steps = 5e4
tune_range = 10 # GHz
# Construct an odd range of frequencies that is guaranteed to include
# the central resonance frequency.
detuning_range = np.append(np.linspace(-1*tune_range, 0, tune_steps),
                    np.linspace(0, tune_range, tune_steps)[1:])*1e9 # GHz
# Line width of resonances should be v_g * alpha,
# but we don't have convenient access to v_g, therefore
# phase velocity as approximation to group velocity
phase_v = sim_AC_wguide.Eig_values/k_AC
linewidth = phase_v*alpha

freq_min = 5  # GHz
freq_max = 10  # GHz
interp_grid_points = 10000
interp_grid = np.linspace(freq_min, freq_max, interp_grid_points)
interp_values = np.zeros(interp_grid_points)

plt.figure()
```

```
plt.clf()
for AC_i in range(len(alpha)):
    gain_list = np.real(SBS_gain[EM_ival1,EM_ival2,AC_i]/alpha[AC_i]
                *linewidth[AC_i]**2/(linewidth[AC_i]**2 + detuning_range**2))
    freq_list_GHz = np.real(sim_AC_wguide.Eig_values[AC_i] + detuning_range)*1e-9
    plt.plot(freq_list_GHz, gain_list)
    # set up an interpolation for summing all the gain peaks
    interp_spectrum = np.interp(interp_grid, freq_list_GHz, gain_list)
    interp_values += interp_spectrum
plt.plot(interp_grid, interp_values, 'k', linewidth=3, label="Total")
plt.legend(loc=0)
plt.xlim(freq_min,freq_max)
plt.xlabel('Frequency (GHz)')
plt.ylabel('Gain (1/Wm)')
plt.savefig('As2S3-gain_spectra-mode_comps.pdf')
plt.close()

freq_min_zoom = 7.5
freq_max_zoom = 8.6
# freq_min_zoom = freq_min
# freq_max_zoom = freq_max
interp_values = np.zeros(interp_grid_points)
interp_values_PE = np.zeros(interp_grid_points)
interp_values_MB = np.zeros(interp_grid_points)
plt.figure()
plt.clf()
for AC_i in range(len(alpha)):
    gain_list = np.real(SBS_gain[EM_ival1,EM_ival2,AC_i]/alpha[AC_i]
                *linewidth[AC_i]**2/(linewidth[AC_i]**2 + detuning_range**2))
    freq_list_GHz = np.real(sim_AC_wguide.Eig_values[AC_i] + detuning_range)*1e-9
    interp_spectrum = np.interp(interp_grid, freq_list_GHz, gain_list)
    interp_values += interp_spectrum

    gain_list_PE = np.real(SBS_gain_PE[EM_ival1,EM_ival2,AC_i]/alpha[AC_i]
                *linewidth[AC_i]**2/(linewidth[AC_i]**2 + detuning_range**2))
    interp_spectrum_PE = np.interp(interp_grid, freq_list_GHz, gain_list_PE)
    interp_values_PE += interp_spectrum_PE

    gain_list_MB = np.real(SBS_gain_MB[EM_ival1,EM_ival2,AC_i]/alpha[AC_i]
                *linewidth[AC_i]**2/(linewidth[AC_i]**2 + detuning_range**2))
    interp_spectrum_MB = np.interp(interp_grid, freq_list_GHz, gain_list_MB)
    interp_values_MB += interp_spectrum_MB
plt.plot(interp_grid, interp_values, 'k', linewidth=3, label="Total")
plt.plot(interp_grid, interp_values_PE, 'r', linewidth=3, label="PE")
plt.plot(interp_grid, interp_values_MB, 'g', linewidth=3, label="MB")
plt.legend(loc=0)
plt.xlim(freq_min_zoom,freq_max_zoom)
plt.xlabel('Frequency (GHz)')
plt.ylabel('Gain (1/Wm)')
plt.savefig('As2S3-gain_spectra-MB_PE_comps.pdf')
plt.close()


plt.figure()
plt.clf()
plt.plot(interp_grid, 20*np.log10(abs(interp_values)), 'k', linewidth=3, label="Total")
plt.plot(interp_grid, 20*np.log10(abs(interp_values_PE)), 'r', linewidth=3, label="PE")
plt.plot(interp_grid, 20*np.log10(abs(interp_values_MB)), 'g', linewidth=3, label="MB")
```

```
plt.legend(loc=0)
plt.xlim(freq_min_zoom,freq_max_zoom)
plt.xlabel('Frequency (GHz)')
plt.ylabel('Gain (dB)')
plt.savefig('As2S3-gain_spectra-MB_PE_comps-dB.pdf')
plt.close()
```

Silica Nanowire Example

```
plt.close()
""" Calculate the backward SBS gain spectra of a
    silicon waveguide surrounded in air.

    Show how to save simulation objects (eg EM mode calcs)
    to expedite the process of altering later parts of
    simulations.
"""

import time
import datetime
import numpy as np
import sys
sys.path.append("../backend/")
import matplotlib
matplotlib.use('pdf')
import matplotlib.pyplot as plt

import materials
import objects
import mode_calcs
import integration
import plotting
from fortran import NumBAT


# Geometric Parameters - all in nm.
wl_nm = 1550
unitcell_x = 5*wl_nm
unitcell_y = unitcell_x
inc_a_x = 550
inc_a_y = inc_a_x
inc_shape = 'circular'

# Optical Parameters
n_inc_a = 1.44
num_EM_modes = 20
num_AC_modes = 40
EM_ival1=0
EM_ival2=EM_ival1
AC_ival='All'

# # Silca
# s = 2200  # kg/m3
# E = 7.3e10
# v = 0.17
# c_11, c_12, c_44 = materials.isotropic_stiffness(E, v)
```

```
# p_11 = 0.121; p_12 = 0.270; p_44 = -0.075
# eta_11 = 1.6e-3 ; eta_12 = 1.29e-3 ; eta_44 = 0.16e-3  # Pa s


# Silca - Laude AIP Advances 2013
s = 2203  # kg/m3
c_11 = 78e9; c_12 = 16e9; c_44 = 31e9
p_11 = 0.12; p_12 = 0.270; p_44 = -0.073
eta_11 = 1.6e-3 ; eta_12 = 1.29e-3 ; eta_44 = 0.16e-3  # Pa s

inc_a_AC_props = [s, c_11, c_12, c_44, p_11, p_12, p_44,
                  eta_11, eta_12, eta_44]

# Use all specified parameters to create a waveguide object.
wguide = objects.Struct(unitcell_x,inc_a_x,unitcell_y,inc_a_y,inc_shape,
                        bkg_material=materials.Material(1.0 + 0.0j),
                        inc_a_material=materials.Material(n_inc_a),
                        loss=False, inc_a_AC=inc_a_AC_props,
                        lc_bkg=3, lc2=2000.0, lc3=20.0)

# Expected effective index of fundamental guided mode.
n_eff=1.4

# Calculate Electromagnetic Modes
sim_EM_wguide = wguide.calc_EM_modes(wl_nm, num_EM_modes, n_eff=n_eff)
# np.savez('wguide_data', sim_EM_wguide=sim_EM_wguide)
# npzfile = np.load('wguide_data.npz')
# sim_EM_wguide = npzfile['sim_EM_wguide'].tolist()
# plotting.plt_mode_fields(sim_EM_wguide, xlim=0.4, ylim=0.4, EM_AC='EM', add_name='NW')
# plotting.plt_mode_fields(sim_EM_wguide, EM_AC='EM', add_name='NW')

# Print the wavevectors of EM modes.
print 'k_z of EM modes \n', np.round(np.real(sim_EM_wguide.Eig_values), 4)

# Calculate the EM effective index of the waveguide.
n_eff_sim = np.real(sim_EM_wguide.Eig_values*((wl_nm*1e-9)/(2.*np.pi)))
print "n_eff = ", np.round(n_eff_sim, 4)

k_AC = 2*np.real(sim_EM_wguide.Eig_values[0])

shift_Hz = 4e9

# Calculate Acoustic Modes
sim_AC_wguide = wguide.calc_AC_modes(wl_nm, num_AC_modes, k_AC=k_AC,
    EM_sim=sim_EM_wguide, shift_Hz=shift_Hz)
# np.savez('wguide_data_AC', sim_AC_wguide=sim_AC_wguide)
# npzfile = np.load('wguide_data_AC.npz')
# sim_AC_wguide = npzfile['sim_AC_wguide'].tolist()
# plotting.plt_mode_fields(sim_AC_wguide, EM_AC='AC', add_name='NW')

# Print the frequencies of AC modes.
print 'Freq of AC modes (GHz) \n', np.round(np.real(sim_AC_wguide.Eig_values)*1e-9, 4)

# Do not calculate the acoustic loss from our fields, but instead set a
# predetirmined Q factor. (Useful for instance when replicating others results).
set_q_factor = 1000.

# Calculate interaction integrals and SBS gain for PE and MB effects combined,
```

```
# as well as just for PE, and just for MB. Also calculate acoustic loss alpha.
SBS_gain, SBS_gain_PE, SBS_gain_MB, alpha = integration.gain_and_qs(
    sim_EM_wguide, sim_AC_wguide, k_AC,
    EM_ival1=EM_ival1, EM_ival2=EM_ival2, AC_ival=AC_ival, fixed_Q=set_q_factor)
# np.savez('wguide_data_AC_gain', SBS_gain=SBS_gain, alpha=alpha)
# npzfile = np.load('wguide_data_AC_gain.npz')
# SBS_gain = npzfile['SBS_gain']
# alpha = npzfile['alpha']


# Construct the SBS gain spectrum, built up
# from Lorentzian peaks of the individual modes.
tune_steps = 5e4
tune_range = 10 # GHz
# Construct an odd range of frequencies that is guaranteed to include
# the central resonance frequency.
detuning_range = np.append(np.linspace(-1*tune_range, 0, tune_steps),
                  np.linspace(0, tune_range, tune_steps)[1:])*1e9 # GHz
# Line width of resonances should be v_g * alpha,
# but we don't have convenient access to v_g, therefore
# phase velocity as approximation to group velocity
phase_v = sim_AC_wguide.Eig_values/k_AC
linewidth = phase_v*alpha

freq_min = 0  # GHz
freq_max = 12  # GHz
interp_grid_points = 10000
interp_grid = np.linspace(freq_min, freq_max, interp_grid_points)
interp_values = np.zeros(interp_grid_points)

freq_threshold = 5  # GHz

plt.figure()
plt.clf()
for AC_i in range(len(alpha)):
    if np.real(sim_AC_wguide.Eig_values[AC_i])*1e-9 > freq_threshold:
        # print SBS_gain[EM_ival1,EM_ival2,AC_i]
        # print alpha[AC_i]
        gain_list = np.real(SBS_gain[EM_ival1,EM_ival2,AC_i]/alpha[AC_i]
                     *linewidth[AC_i]**2/(linewidth[AC_i]**2 + detuning_range**2))
        freq_list_GHz = np.real(sim_AC_wguide.Eig_values[AC_i] + detuning_range)*1e-9
        plt.plot(freq_list_GHz, gain_list)
        # set up an interpolation for summing all the gain peaks
        interp_spectrum = np.interp(interp_grid, freq_list_GHz, gain_list)
        interp_values += interp_spectrum
plt.plot(interp_grid, interp_values, 'k', linewidth=3, label="Total")
plt.legend(loc=0)
plt.xlim(freq_min,freq_max)
plt.xlabel('Frequency (GHz)')
plt.ylabel('Gain (1/Wm)')
plt.savefig('gain_spectra-mode_comps.pdf')
plt.close()

freq_min_zoom = freq_min
freq_max_zoom = freq_max
interp_values = np.zeros(interp_grid_points)
interp_values_PE = np.zeros(interp_grid_points)
interp_values_MB = np.zeros(interp_grid_points)
```

```
plt.figure()
plt.clf()
for AC_i in range(len(alpha)):
    gain_list = np.real(SBS_gain[EM_ival1,EM_ival2,AC_i]/alpha[AC_i]
                *linewidth[AC_i]**2/(linewidth[AC_i]**2 + detuning_range**2))
    freq_list_GHz = np.real(sim_AC_wguide.Eig_values[AC_i] + detuning_range)*1e-9
    interp_spectrum = np.interp(interp_grid, freq_list_GHz, gain_list)
    interp_values += interp_spectrum

    gain_list_PE = np.real(SBS_gain_PE[EM_ival1,EM_ival2,AC_i]/alpha[AC_i]
                *linewidth[AC_i]**2/(linewidth[AC_i]**2 + detuning_range**2))
    interp_spectrum_PE = np.interp(interp_grid, freq_list_GHz, gain_list_PE)
    interp_values_PE += interp_spectrum_PE

    gain_list_MB = np.real(SBS_gain_MB[EM_ival1,EM_ival2,AC_i]/alpha[AC_i]
                *linewidth[AC_i]**2/(linewidth[AC_i]**2 + detuning_range**2))
    interp_spectrum_MB = np.interp(interp_grid, freq_list_GHz, gain_list_MB)
    interp_values_MB += interp_spectrum_MB
plt.plot(interp_grid, interp_values, 'k', linewidth=3, label="Total")
plt.plot(interp_grid, interp_values_PE, 'r', linewidth=3, label="PE")
plt.plot(interp_grid, interp_values_MB, 'g', linewidth=3, label="MB")
plt.legend(loc=0)
plt.xlim(freq_min_zoom,freq_max_zoom)
plt.xlabel('Frequency (GHz)')
plt.ylabel('Gain (1/Wm)')
plt.savefig('gain_spectra-MB_PE_comps.pdf')
plt.close()
```

CHAPTER

FOUR

PYTHON BACKEND

objects module

objects.py is a subroutine of NumBAT. It contains the Struct objects that represent the structure being simulated.

class objects.**NumBAT**
Bases: object

class objects.**Struct**(*unitcell_x*, *inc_a_x*, *unitcell_y=None*, *inc_a_y=None*, *inc_shape='rectangular'*, *slab_a_x=None*, *slab_a_y=None*, *slab_b_x=None*, *slab_b_y=None*, *coating_y=None*, *inc_b_x=None*, *inc_b_y=None*, *two_inc_sep=None*, *incs_y_offset=None*, *bkg_material=<materials.Material object>*, *inc_a_material=<materials.Material object>*, *inc_b_material=<materials.Material object>*, *slab_a_material=<materials.Material object>*, *slab_a_bkg_material=<materials.Material object>*, *slab_b_material=<materials.Material object>*, *slab_b_bkg_material=<materials.Material object>*, *coating_material=<materials.Material object>*, *inc_c_x=None*, *inc_d_x=None*, *inc_e_x=None*, *inc_f_x=None*, *inc_g_x=None*, *inc_h_x=None*, *inc_i_x=None*, *inc_j_x=None*, *inc_k_x=None*, *inc_l_x=None*, *inc_m_x=None*, *inc_n_x=None*, *inc_o_x=None*, *inc_c_material=<materials.Material object>*, *inc_d_material=<materials.Material object>*, *inc_e_material=<materials.Material object>*, *inc_f_material=<materials.Material object>*, *inc_g_material=<materials.Material object>*, *inc_h_material=<materials.Material object>*, *inc_i_material=<materials.Material object>*, *inc_j_material=<materials.Material object>*, *inc_k_material=<materials.Material object>*, *inc_l_material=<materials.Material object>*, *inc_m_material=<materials.Material object>*, *inc_n_material=<materials.Material object>*, *inc_o_material=<materials.Material object>*, *inc_c_AC=None*, *inc_d_AC=None*, *inc_e_AC=None*, *inc_f_AC=None*, *inc_g_AC=None*, *inc_h_AC=None*, *inc_i_AC=None*, *inc_j_AC=None*, *inc_k_AC=None*, *inc_l_AC=None*, *inc_m_AC=None*, *inc_n_AC=None*, *inc_o_AC=None*, *loss=True*, *bkg_AC=None*, *inc_a_AC=None*, *slab_a_AC=None*, *slab_a_bkg_AC=None*, *slab_b_AC=None*, *slab_b_bkg_AC=None*, *make_mesh_now=True*, *force_mesh=True*, *mesh_file='NEED_FILE.mail'*, *check_msh=False*, *lc_bkg=0.09*, *lc2=1.0*, *lc3=1.0*, *lc4=1.0*, *lc5=1.0*, *lc6=1.0*, *plotting_fields=False*, *plot_real=1*, *plot_imag=0*, *plot_abs=0*, *plot_field_conc=False*)

Bases: object

Represents a structured layer.

Parameters

- **unitcell_x** (*float*) – The horizontal period of the unit cell in nanometers.
- **inc_a_x** (*float*) – The horizontal diameter of the inclusion in nm.

Keyword Arguments

- **unitcell_y** (*float*) – The vertical period of the unit cell in nanometers. If None, unitcell_y = unitcell_x.
- **inc_a_y** (*float*) – The vertical diameter of the inclusion in nm.
- **inc_shape** (*str*) – Shape of inclusions that have template mesh, currently: 'circular', 'rectangular'. Rectangular is default.
- **slab_a_x** (*float*) – The horizontal diameter in nm of the slab directly below the inclusion.
- **slab_a_y** (*float*) – The vertical diameter in nm of the slab directly below the inclusion.
- **slab_b_x** (*float*) – The horizontal diameter in nm of the slab separated from the inclusion by slab_a.

- **slab_b_y** (*float*) – The vertical diameter in nm of the slab separated from the inclusion by slab_a.
- **two_inc_sep** (*float*) – Separation between edges of inclusions in nm.
- **incs_y_offset** (*float*) – Vertical offset between centers of inclusions in nm.
- **coating_y** (*float*) – The thickness of any coating layer around the inclusion.
- **background** – A :Material: instance for the background medium.
- **inc_a_material** – A :Material: instance for the
- **inc_b_material** – A :Material: instance for the
- **slab_a_material** – A :Material: instance for the
- **slab_a_bkg_material** – A :Material: instance for the
- **slab_b_material** – A :Material: instance for the
- **slab_b_bkg_material** – A :Material: instance for the
- **coating_material** – A :Material: instance for the
- **loss** (*bool*) – If False, Im(n) = 0, if True n as in :Material: instance.
- **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 pre-made mesh give its name including .mail if 2D_array (eg. 600_60.mail), or .txt if 1D_array. It must be located in backend/fortran/msh/
- **lc_bkg** (*float*) – Length constant of meshing of background medium (smaller = finer mesh)
- **lc2** (*float*) – factor by which lc_bkg will be reduced on inclusion surfaces; lc_surface = cl_bkg / lc2.
- **lc3-6'** (*float*) – factor by which lc_bkg will be reduced at center of inclusions.
- **plotting_fields** (*bool*) – Unless set to true field data deleted. Also plots modes (ie. FEM solutions) in gmsh format. Plots epsilon*|E|^2 & choice of real/imag/abs of x,y,z components & field vectors. Fields are saved as gmsh files, but can be converted by running the .geo file found in Bloch_fields/PNG/
- **plot_real** (*bool*) – Choose to plot real part of modal fields.
- **plot_imag** (*bool*) – Choose to plot imaginary part of modal fields.
- **plot_abs** (*bool*) – Choose to plot absolute value of modal fields.

calc_AC_modes(*wl_nm*, *num_modes*, *k_AC*, *shift_Hz=None*, *EM_sim=None*, ***args*)

Run a simulation to find the Struct's acoustic modes.

Parameters

- **wl_nm** (*float*) – Wavelength of AC wave in vacuum.
- **num_modes** (*int*) – Number of AC modes to solve for.

Keyword Arguments

- **shift_Hz** (*float*) – Guesstimated frequency of modes, will be origin of FEM search. NumBAT will make an educated guess if shift_Hz=None. (Technically the shift and invert parameter).
- **EM_sim (** – Simmo: object): Typically an acoustic simulation follows on from an optical one. Supply the EM :Simmo: object so the AC FEM mesh can be constructed from this. This is done by removing vacuum regions.

Returns Simmo: object

calc_EM_modes(*wl_nm*, *num_modes*, *n_eff*, ***args*)
Run a simulation to find the Struct's EM modes.

Parameters

- **wl_nm** (*float*) – Wavelength of EM wave in vacuum.
- **num_modes** (*int*) – Number of EM modes to solve for.
- **n_eff** (*float*) – Guesstimated effective index of fundamental mode, will be origin of FEM search.

Returns Simmo: object

make_mesh()
Take the parameters specified in python and make a Gmsh FEM mesh. Creates a .geo and .msh file, then uses Fortran conv_gmsh routine to convert .msh into .mail, which is used in NumBAT FEM routine.

objects.**dec_float_str**(*dec_float*)
Convert float with decimal point into string with '_' in place of '.'

materials module

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

Copyright (C) 2016 Bjorn Sturmberg, Kokou Dossou

class materials.**Material**(*n*)
Bases: object

Represents a material with a refractive index n.

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

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

Currently included materials are;

Semiconductors	Metals	Transparent oxides
Si_c	Au	TiO2
Si_a	Au_Palik	TiO2_anatase
SiO2	Ag	ITO
CuO	Ag_Palik	ZnO
CdTe	Cu	SnO2
FeS2	Cu_Palik	FTO_Wenger
Zn3P2		FTO_Wengerk5

Continued on next page

Table 4.1 – continued from previous page

AlGaAs		
Al2O3		
Al2O3_PV		
GaAs		
InGaAs	**Drude**	**Other**
Si3N4	Au_drude	Air
MgF2		H2O
InP		Glass
InAs		Spiro
GaP		Spiro_nk
Ge		
AlN		
GaN		
MoO3		
ZnS		
AlN_PV		
		Experimental incl.
		CH3NH3PbI3
		Sb2S3
		Sb2S3_ANU2014
		Sb2S3_ANU2015
		GO_2014
		GO_2015
		rGO_2015
		SiON_Low
		SiON_High
		Low_Fe_Glass
		Perovskite_00
		Perovskite
		Perovskite_b2b
		Ge_Doped

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

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

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

materials.**isotropic_stiffness**(*E*, *v*)
Calculate the stiffness matrix components of isotropic materials, given the two free parameters: E: Youngs_modulus v: Poisson_ratio

Ref: http://www.efunda.com/formulae/solid_mechanics/mat_mechanics/hooke_isotropic.cfm

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

mode_calcs module

mode_calcs.py is a subroutine of NumBAT that contains methods to calculate the EM and Acoustic modes of a structure.

class mode_calcs.**NumBAT**
Bases: object

class mode_calcs.**Simmo**(*structure*, *wl_nm*, *num_modes=20*, *n_eff=None*, *shift_Hz=None*, *k_AC=None*, *EM_sim=None*)
Bases: object

Calculates the modes of a :Struc: object at a wavelength of wl_nm.

calc_AC_modes()
Run a Fortran FEM calculation to find the acoustic modes.

Returns a :Simmo: object that now has these key values:

Eig_values: a 1d array of Eigenvalues (frequencies) in [1/s]

sol1: the associated Eigenvectors, ie. the fields, stored as [field comp, node nu on element, Eig value, el nu]

calc_EM_modes()
Run a Fortran FEM calculation to find the optical modes.

Returns a :Simmo: object that now has these key values:

Eig_values: a 1d array of Eigenvalues (propagation constants) in [1/m]

sol1: the associated Eigenvectors, ie. the fields, stored as [field comp, node nu on element, Eig value, el nu]

integration module

mode_calcs.py is a subroutine of NumBAT that contains methods to calculate the EM and Acoustic modes of a structure.

class integration.**NumBAT**
Bases: object

integration.**gain_and_qs**(*sim_EM_wguide*, *sim_AC_wguide*, *k_AC*, *EM_ival1=0*, *EM_ival2=0*, *AC_ival=0*, *fixed_Q=None*, *typ_select_out=None*)
Calculate interaction integrals and SBS gain.

Implements Eqs. 33, 41, 45 of Wolff et al. PRA 92, 013836 (2015) doi/10.1103/PhysRevA.92.013836 These are for Q_photoelastic, Q_moving_boundary, and the Acoustic loss "alpha" respectively.

Parameters

- **(** (*sim_AC_wguide*) – Simmo: object): Contains all info on EM modes
- **(** – Simmo: object): Contains all info on AC modes
- **k_AC** (*float*) – Propagation constant of acoustic modes.

Keyword Arguments

- **EM_ival1** (*int/string*) – Specify mode number of EM mode 1 (pump mode) to calculate interactions for. Numbering is python index so runs from 0 to num_EM_modes-1, with 0 being fundamental mode (largest prop constant). Can also set to 'All' to include all modes.
- **EM_ival2** (*int/string*) – Specify mode number of EM mode 2 (stokes mode) to calculate interactions for. Numbering is python index so runs from 0 to num_EM_modes-1, with 0 being fundamental mode (largest prop constant). Can also set to 'All' to include all modes.
- **AC_ival** (*int/string*) – Specify mode number of AC mode to calculate interactions for. Numbering is python index so runs from 0 to num_AC_modes-1, with 0 being fundamental mode (largest prop constant). Can also set to 'All' to include all modes.
- **fixed_Q** (*int*) – Specify a fixed Q-factor for the AC modes, rather than calculating the acoustic loss (alpha).

Returns **SBS_gain** – The SBS gain including both photoelastic and moving boundary contributions. SBS_gain_PE (num_modes_EM,num_modes_EM,num_modes_AC): The SBS gain for only the photoelastic effect. SBS_gain_MB (num_modes_EM,num_modes_EM,num_modes_AC): The SBS gain for only the moving boundary effect. alpha (num_modes_AC): The acoustic loss for each mode.

Return type num_modes_EM,num_modes_EM,num_modes_AC

integration.**symmetries**(*sim_wguide*, *n_points=10*)

Plot EM mode fields.

Parameters **sim_wguide** – A :Struct: instance that has had calc_modes calculated

Keyword Arguments **n_points** (*int*) – The number of points across unitcell to interpolate the field onto.

plotting module

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

Copyright (C) 2016 Bjorn Sturmberg

plotting.**plot_msh**(*x_arr*, *add_name=''*)

Plot EM mode fields.

Parameters **sim_wguide** – A :Struct: instance that has had calc_modes calculated

Keyword Arguments **n_points** (*int*) – The number of points across unitcell to interpolate the field onto.

plotting.**plt_mode_fields**(*sim_wguide*, *n_points=500*, *quiver_steps=50*, *xlim=None*, *ylim=None*, *EM_AC='EM'*, *pdf_png='png'*, *add_name=''*)

Plot EM mode fields. NOTE: z component of EM field needs comes scaled by 1/(i beta), which must be reintroduced!

Args: sim_wguide : A :Struct: instance that has had calc_modes calculated

Keyword Args: n_points (int): The number of points across unitcell to interpolate the field onto.

xlim (float): Limit plotted xrange to xlim:(1-xlim) of unitcell

ylim (float): Limit plotted yrange to ylim:(1-ylim) of unitcell

plotting.**zeros_int_str**(*zero_int*)

Convert integer into string with '0' in place of ' '.

CHAPTER

FIVE

FORTRAN BACKENDS

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

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

2D FEM Mode Solver

2D Mesh

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

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

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

FEM Errors

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

The first of these is

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

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

The second error is

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

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

2D FEM Mode Solver

2D Mesh

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

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

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

FEM Errors

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

The first of these is

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

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

The second error is

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

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

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

i

m

o

p

Symbols

C

D

G

I

M

N

O

P

S

Z