A_FMM Documentation

Marco Passoni

Feb 27, 2024

CONTENTS:

Python implementation the Aperiodic-Fourier Modal Method. A fully vectorial method for solving Maxwell equations that combines a Fourier-based mode solver and a scattering matrix recursion algorithm to model full 3D structures. This approach is well suited to calculate modes, transmission, reflection, scattering and absorption of multi-layered structures. Moreover, support for Bloch modes of periodic structures allows for the simulation of photonic crystals or waveguide Bragg gratings.

CHAPTER

ONE

INTRODUCTION

This is the draft documentation for the A_FMM package. It is a work in progress. Any suggestion for improvements is welcome.

1.1 Installation

You can install A_FMM directly from pypi by running:

```
pip install A_FMM
```

1.2 Method Overview

The A-FMM method as we know it appeared in the mid 2005[1]. It builds on the Fourier modal method for crossed gratings[2], which combines Fourier methods with the well-known scattering matrix algorithm[3]. The novelty of the A-FMM is the addition of a coordinate transformation to map the unit cell of the grating to the full 2 space, thus allowing the modeling of aperiodic structures. The transformation exists in real and complex variants[4], where the latter behaves as a PML and thus allows for the treatment of structure with high scattering loss. The implementation in this paper complements these core functionalities with methods for calculating Bloch modes from the scattering matrix of the unit cell[5] and a custom method for calculating Poynting vectors and energy flows[6].

This implementation aims at providing easy access to the A-FMM method to as many people as possible. Therefore, it is implemented in Python3, making this tool easy to install and portable across different platforms. It makes use of classes for a simple and modular structure (figure below). Being heavily based on numpy[7] and scipy[8] libraries, it uses well tested numerical routines known for their efficiency, stability and scalability.

Moreover, this implementation was battle tasted on HPC system and was employed during research activities leading

[1] Hugonin, J. P., et al. "Fourier modal methods for modeling optical dielectric waveguides." Optical and quantum electronics 37.1 (2005): 107-119.

[2] Li, Lifeng. "New formulation of the Fourier modal method for crossed surface-relief gratings." JOSA A 14.10 (1997): 2758-2767.

[3] Li, Lifeng. "Formulation and comparison of two recursive matrix algorithms for modeling layered diffraction gratings." JOSA A 13.5 (1996): 1024-1035.

[4] Hugonin, Jean Paul, and Philippe Lalanne. "Perfectly matched layers as nonlinear coordinate transforms: a generalized formalization." JOSA A 22.9 (2005): 1844-1849.

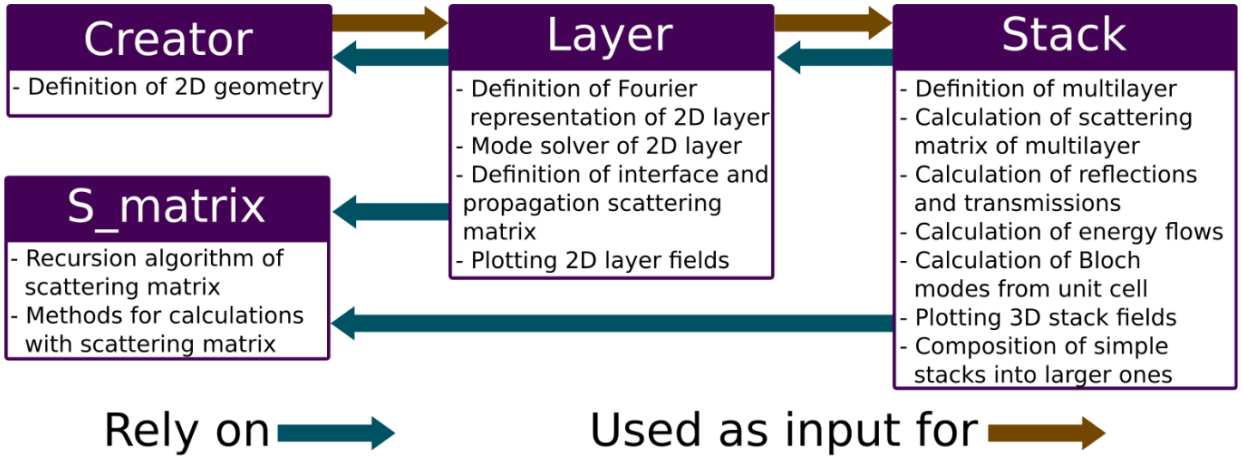
[5] Li, Zhi-Yuan, and Lan-Lan Lin. "Photonic band structures solved by a plane-wave-based transfer-matrix method." Physical Review E 67.4 (2003): 046607.

[6] Passoni, Marco. "Theoretical study of integrated grating structures for Silicon Photonics."

[7] Harris, Charles R., et al. "Array programming with NumPy." Nature 585.7825 (2020): 357-362.

[8] Virtanen, Pauli, et al. "SciPy 1.0: fundamental algorithms for scientific computing in Python." Nature methods 17.3 (2020): 261-272.

to published papers[9][10][11].



Structure of the A-FMM code. Blocks represent Python classes and arrows the relations between them.

[9] Passoni, Marco, et al. "Optimizing band-edge slow light in silicon-on-insulator waveguide gratings." Optics Express 26.7 (2018): 8470-8478.
[10] Passoni, Marco, et al. "Slow light with interleaved pn junction to enhance performance of integrated Mach-Zehnder silicon modulators." Nanophotonics 8.9 (2019): 1485-1494.
[11] Fornasari, Lucia, et al. "Angular dependence and absorption properties of the anapole mode of Si nano-disks." Journal of Applied Physics 129.2 (2021): 023102.

CHAPTER

TWO

EXAMPLES

2.1 Mode Solver

2.1.1 1D mode solver

This section provides an example of mode solving for a 1D structure. This example is aslo used to illustrate the use of coordinate transformation in its real and complex variant.

Importing packages

[1]:
```
import numpy as np
import matplotlib.pyplot as plt

import A_FMM
```

Defining structure

The definition of a layer uses two classes. The class Creator is responsible for definiiton of the geometry, while the class Layer is responsible for defining the layer object used for simulation. The creator object as a few geometry already inplemented. Please see documentaiton for details.

The next code block define a single layer contained the gemoetry of a 1D slab, apply real coordinate transformation and plot the epsilon profile reconstructed from fourier transform.

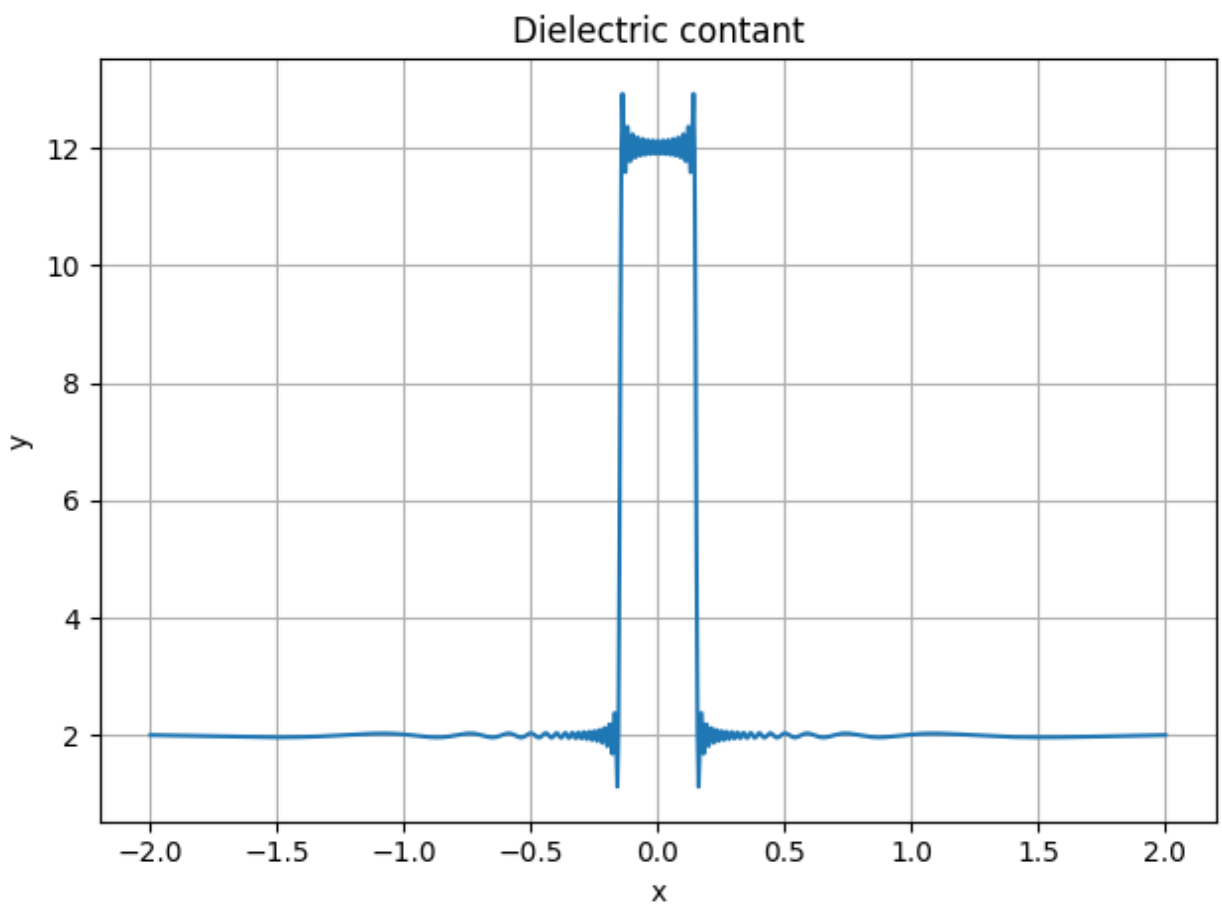
[2]:
```
cr = A_FMM.Creator()
cr.slab(12.0, 2.0, 2.0, 0.3)
lay = A_FMM.Layer(Nx=50, Ny=0, creator=cr)
lay.transform(ex=0.5)

x = np.linspace(-2.0, 2.0, 1001)
eps = lay.calculate_epsilon(x)
plt.plot(x, np.squeeze(eps['eps']))
plt.xlabel('x'), plt.ylabel('y'), plt.title('Dielectric contant'), plt.grid()
plt.tight_layout()
```

```
/home/marco/Documents/MyPrograms/A_FMM/test_venv/lib/python3.10/site-packages/matplotlib/
↪cbook.py:1699: ComplexWarning: Casting complex values to real discards the imaginary␣
↪part
  return math.isfinite(val)
```

(continues on next page)

(continued from previous page)

```
/home/marco/Documents/MyPrograms/A_FMM/test_venv/lib/python3.10/site-packages/matplotlib/
↪cbook.py:1345: ComplexWarning: Casting complex values to real discards the imaginary␣
↪part
  return np.asarray(x, float)
```



Effective index and field calculation

The next code blocks calculates the modes of the layer, retrieves the effective index and mode profile of the first 4 modes, and plots them.

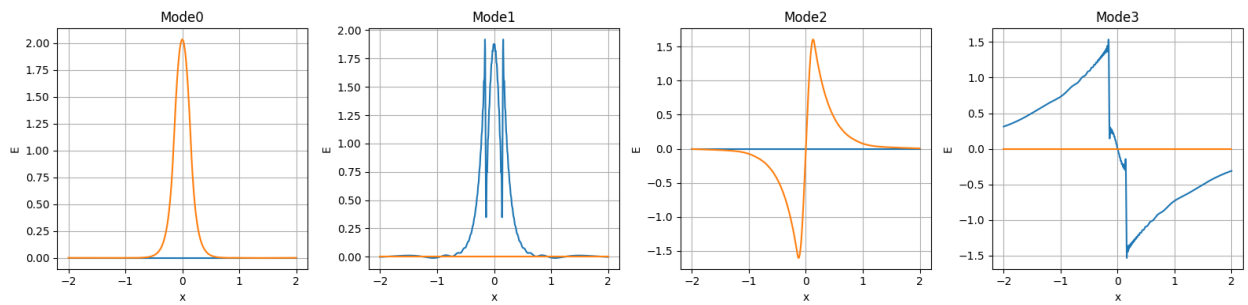
[3]:
```
lay.mode(1.0/1.55)
index = lay.get_index()
fig, ax = plt.subplots(1,4, figsize=(16, 4))
for i in range(4):
    print(f'Mode{i} effective index: {index[i]}')
    u = lay.create_input({i:1.0})
    field = lay.calculate_field(u, x=x)
    Ex = np.squeeze(field['Ex'])
    Ey = np.squeeze(field['Ey'])
    ax[i].set_title(f'Mode{i}')
    ax[i].plot(x, np.real(Ex) + np.imag(Ex), label='Ex')
```

(continues on next page)

(continued from previous page)

```
    ax[i].plot(x, np.real(Ey) + np.imag(Ey), label='Ex')

for a in ax:
    a.set_xlabel('x'), a.set_ylabel('E'), a.grid()

plt.tight_layout()
```

```
Mode0 effective index: (3.035543718671809+0j)
Mode1 effective index: (2.593303259662705 7+3.859641243142317e-17j)
Mode2 effective index: (1.6594616309438428+0j)
Mode3 effective index: (1.426981387566051-2.5630187770423614e-16j)
```



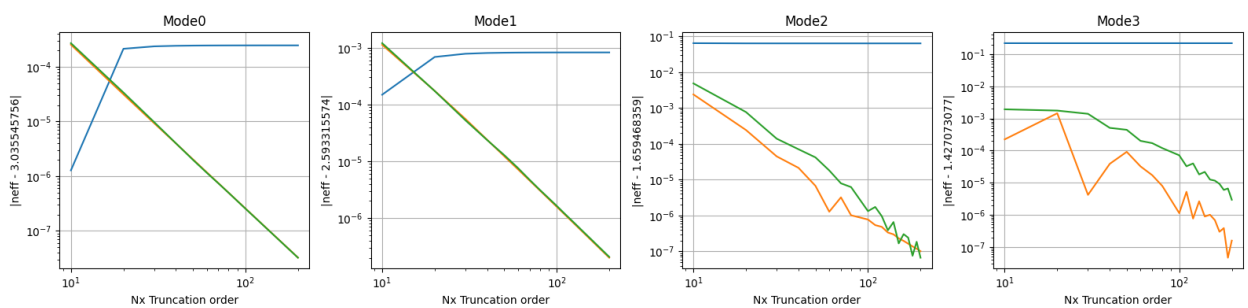
Convergence sweep

```
[4]: Nx_list = [10,20,30,40,50, 60, 70, 80 ,100, 110, 120, 130, 140, 150, 160, 170, 180, 190,␣
↪200]
Neff_no_trasform = {i : [] for i in range(4)}
Neff_re_trasform = {i : [] for i in range(4)}
Neff_cm_trasform = {i : [] for i in range(4)}
Nett_target = {
    0: 3.035545756,
    1: 2.593315574,
    2: 1.659468359,
    3: 1.427073077,
}
for Nx in Nx_list:
    lay = A_FMM.Layer(Nx=Nx, Ny=0, creator=cr)
    lay.mode(1.0/1.55)
    for i, li in Neff_no_trasform.items():
        li.append(lay.get_index()[i])

    lay.transform(ex=0.5)
    lay.mode(1.0/1.55)
    for i, li in Neff_re_trasform.items():
        li.append(lay.get_index()[i])

    lay.transform(ex=0.5, complex_transform=True)
    lay.mode(1.0/1.55)
    index = np.asarray(lay.get_index())
    for i, li in Neff_cm_trasform.items():
```

(continues on next page)

(continued from previous page)

```
        order = np.argsort([abs(_ - Nett_target[i]) for _ in index])
        li.append(index[order[0]])


fig, ax = plt.subplots(1,4, figsize=(16, 4))
for i in range(4):
    ax[i].plot(Nx_list, [abs(_ -Nett_target[i]) for _ in Neff_no_trasform[i]])
    ax[i].plot(Nx_list, [abs(_ -Nett_target[i]) for _ in Neff_re_trasform[i]])
    ax[i].plot(Nx_list, [abs(_ -Nett_target[i]) for _ in Neff_cm_trasform[i]])
    ax[i].set_title(f'Mode{i}')
    ax[i].set_ylabel(f'|neff - {Nett_target[i]}|')
for a in ax:
    a.set_yscale('log'), a.set_xscale('log'), a.set_xlabel('Nx Truncation order'), a.
↪grid()

plt.tight_layout()
```



Effect of coordinate transform on plotting field.

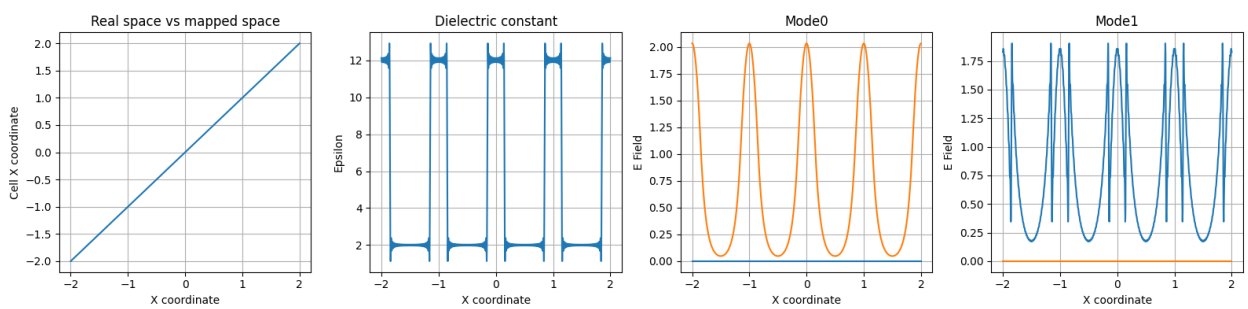
[5]:
```
cr = A_FMM.Creator()
cr.slab(12.0, 2.0, 2.0, 0.3)
lay = A_FMM.Layer(Nx=50, Ny=0, creator=cr)

fig, ax = plt.subplots(1,4, figsize=(16, 4))
x = np.linspace(-2.0, 2.0, 1001)
eps = lay.calculate_epsilon(x)
ax[0].plot(x, np.squeeze(eps['x']))
ax[0].set_ylabel('Cell X coordinate')
ax[0].set_title('Real space vs mapped space')


ax[1].plot(x, np.squeeze(eps['eps']))
ax[1].set_ylabel('Epsilon')
ax[1].set_title('Dielectric constant')

lay.mode(1.0/1.55)
index = lay.get_index()
```

(continues on next page)

(continued from previous page)

```
for i in range(2):
    print(f'Mode{i} effective index: {index[i]}')
    u = lay.create_input({i:1.0})
    field = lay.calculate_field(u, x=x)
    Ex = np.squeeze(field['Ex'])
    Ey = np.squeeze(field['Ey'])
    ax[i+2].set_title(f'Mode{i}')
    ax[i+2].plot(x, np.real(Ex) + np.imag(Ex))
    ax[i+2].plot(x, np.real(Ey) + np.imag(Ey))
    ax[i+2].set_ylabel('E Field')

for a in ax:
    a.set_xlabel('X coordinate'), a.grid()
plt.tight_layout()
```

```
Mode0 effective index: (3.035789650663311+0j)
Mode1 effective index: (2.594132291317708+3.4031786906360965e-15j)
```



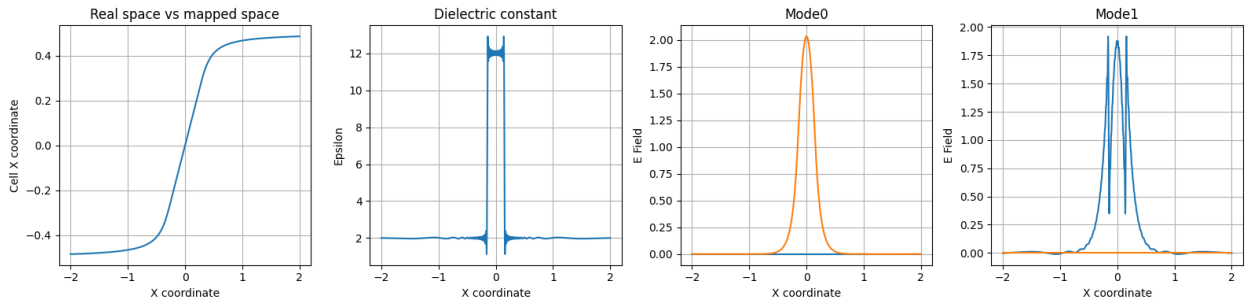
[6]:
```
cr = A_FMM.Creator()
cr.slab(12.0, 2.0, 2.0, 0.3)
lay = A_FMM.Layer(Nx=50, Ny=0, creator=cr)
lay.transform(ex=0.5)

fig, ax = plt.subplots(1,4, figsize=(16, 4))
x = np.linspace(-2.0, 2.0, 1001)
eps = lay.calculate_epsilon(x)
ax[0].plot(x, np.squeeze(eps['x']))
ax[0].set_ylabel('Cell X coordinate')
ax[0].set_title('Real space vs mapped space')


ax[1].plot(x, np.squeeze(eps['eps']))
ax[1].set_ylabel('Epsilon')
ax[1].set_title('Dielectric constant')

lay.mode(1.0/1.55)
index = lay.get_index()
for i in range(2):
    print(f'Mode{i} effective index: {index[i]}')
    u = lay.create_input({i:1.0})
    field = lay.calculate_field(u, x=x)
    Ex = np.squeeze(field['Ex'])
```

(continues on next page)

(continued from previous page)

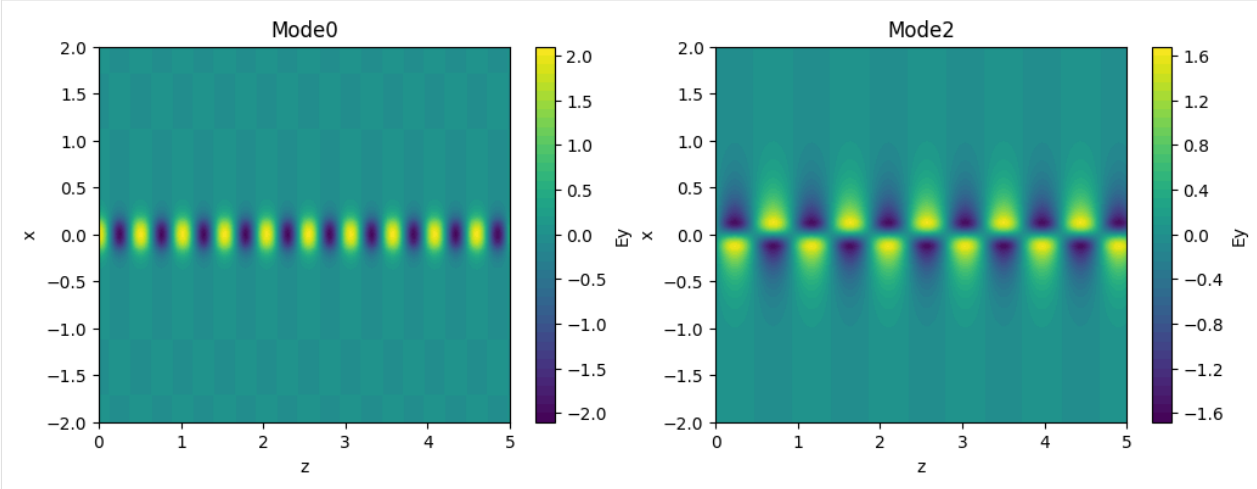
```
    Ey = np.squeeze(field['Ey'])
    ax[i+2].set_title(f'Mode{i}')
    ax[i+2].plot(x, np.real(Ex) + np.imag(Ex))
    ax[i+2].plot(x, np.real(Ey) + np.imag(Ey))
    ax[i+2].set_ylabel('E Field')

for a in ax:
    a.set_xlabel('X coordinate'), a.grid()
plt.tight_layout()
```

```
Mode0 effective index: (3.035543718671809+0j)
Mode1 effective index: (2.5933032596627057+3.859641243142317e-17j)
```



[7]:
```
z = np.linspace(0.0, 5.0, 501)
fig, ax = plt.subplots(1,2, figsize=(12, 4))

u = lay.create_input({0:1.0})
field = lay.calculate_field(u, x=x, z=z)
_ = ax[0].contourf(z,x, np.squeeze(field['Ey']), levels=41)
fig.colorbar(_, ax=ax[0], label='Ey')
ax[0].set_title('Mode0')

u = lay.create_input({2:1.0})
field = lay.calculate_field(u, x=x, z=z)
_=ax[1].contourf(z,x, np.squeeze(field['Ey']), levels=41)
fig.colorbar(_, ax=ax[1], label='Ey')
ax[1].set_title('Mode2')

for a in ax:
    a.set_xlabel('z'), a.set_ylabel('x')
```

```
/home/marco/Documents/MyPrograms/A_FMM/test_venv/lib/python3.10/site-packages/matplotlib/
↪contour.py:1568: ComplexWarning: Casting complex values to real discards the imaginary␣
↪part
  self.zmax = z.max().astype(float)
/home/marco/Documents/MyPrograms/A_FMM/test_venv/lib/python3.10/site-packages/matplotlib/
↪contour.py:1569: ComplexWarning: Casting complex values to real discards the imaginary␣
↪part
  self.zmin = z.min().astype(float)
/home/marco/Documents/MyPrograms/A_FMM/test_venv/lib/python3.10/site-packages/numpy/ma/
↪core.py:2820: ComplexWarning: Casting complex values to real discards the imaginary␣
↪part
  _data = np.array(data, dtype=dtype, copy=copy,
```



2.2 1D Scattering Matrix

2.2.1 1D Bragg Grating

This example illustrates the use of the scattering matrix in a simple 1D case, namely an uniform Bragg grating. Although the simulation is quite simple, it constitutes a good starting point, both theoretically and computationally. Indeed, most to the physics and of the computational methods showed here are the same found in more complex simulations.

The grating in question is built by a periodic repetition of 2 transparent materials (dielectric constant 12.0 and 2.0, respectively). The duty cycle of the grating is 50% and everything is calculated in dimensionless units (or, equivalently, the period of the grating is set to 1).

Import required packages

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import A_FMM
```

Transmission and reflection from a finite multilayer

Building the structure

[2]:
```
import cmath


lay1 = A_FMM.Layer_uniform(0,0,2.0)
lay2 = A_FMM.Layer_uniform(0,0,12.0)

mat = 10*[lay1, lay2] + [lay1]
d   = 10*[0.5, 0.5]   + [0.5]

st = A_FMM.Stack(mat, d)
eps = st.calculate_epsilon(x=[-0.5, 0.5], z = np.linspace(0.0, st.total_length, 1001))
```

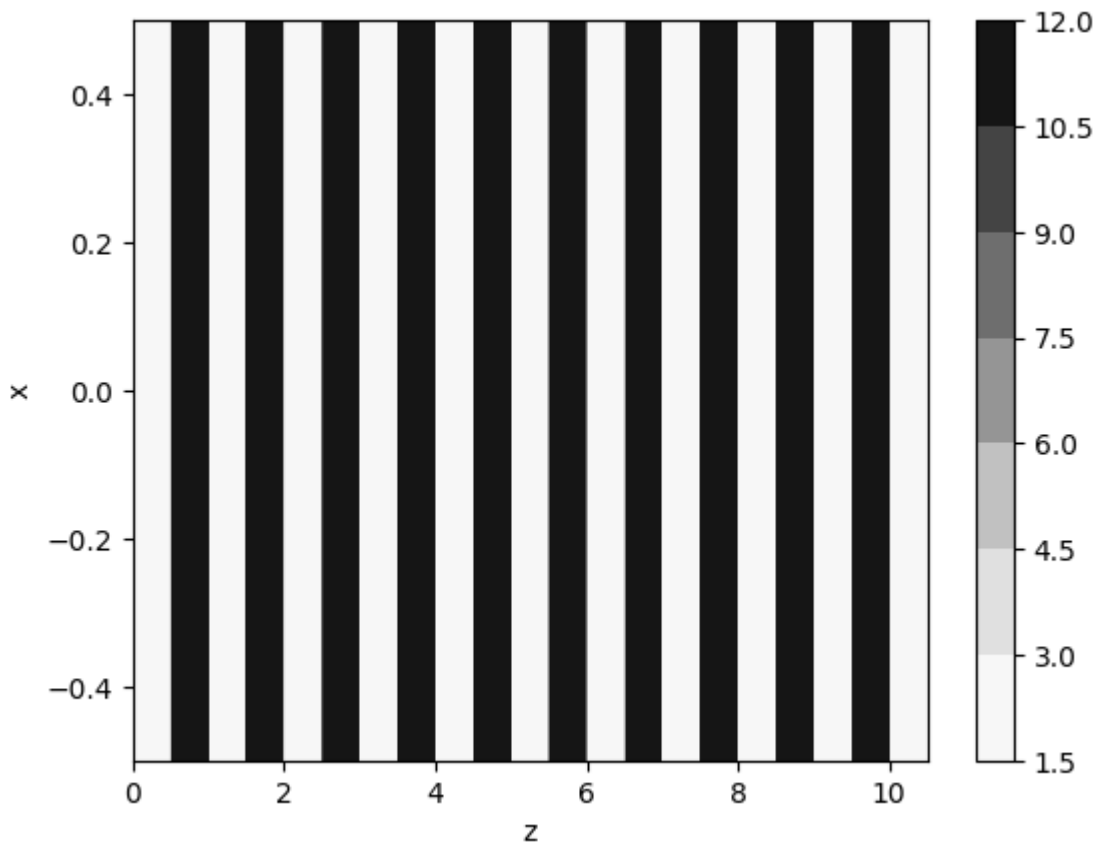
(continues on next page)

(continued from previous page)

```
plt.contourf(np.squeeze(eps['z']), np.squeeze(eps['x']), np.squeeze(eps['eps']), cmap=
↪'Greys')
plt.xlabel('z'), plt.ylabel('x'), plt.colorbar()
```

```
/home/marco/Documents/MyPrograms/A_FMM/test/lib/python3.10/site-packages/matplotlib/
↪contour.py:1568: ComplexWarning: Casting complex values to real discards the imaginary␣
↪part
  self.zmax = z.max().astype(float)
/home/marco/Documents/MyPrograms/A_FMM/test/lib/python3.10/site-packages/matplotlib/
↪contour.py:1569: ComplexWarning: Casting complex values to real discards the imaginary␣
↪part
  self.zmin = z.min().astype(float)
/home/marco/Documents/MyPrograms/A_FMM/test/lib/python3.10/site-packages/numpy/ma/core.
↪py:2820: ComplexWarning: Casting complex values to real discards the imaginary part
  _data = np.array(data, dtype=dtype, copy=copy,
```

[2]:
```
(Text(0.5, 0, 'z'),
 Text(0, 0.5, 'x'),
 <matplotlib.colorbar.Colorbar at 0x7f4551546b30>)
```



Transmission and reflection at normal incidence

[3]:
```
oml = np.linspace(0.0005, 0.6,1200)
DATA = []
```

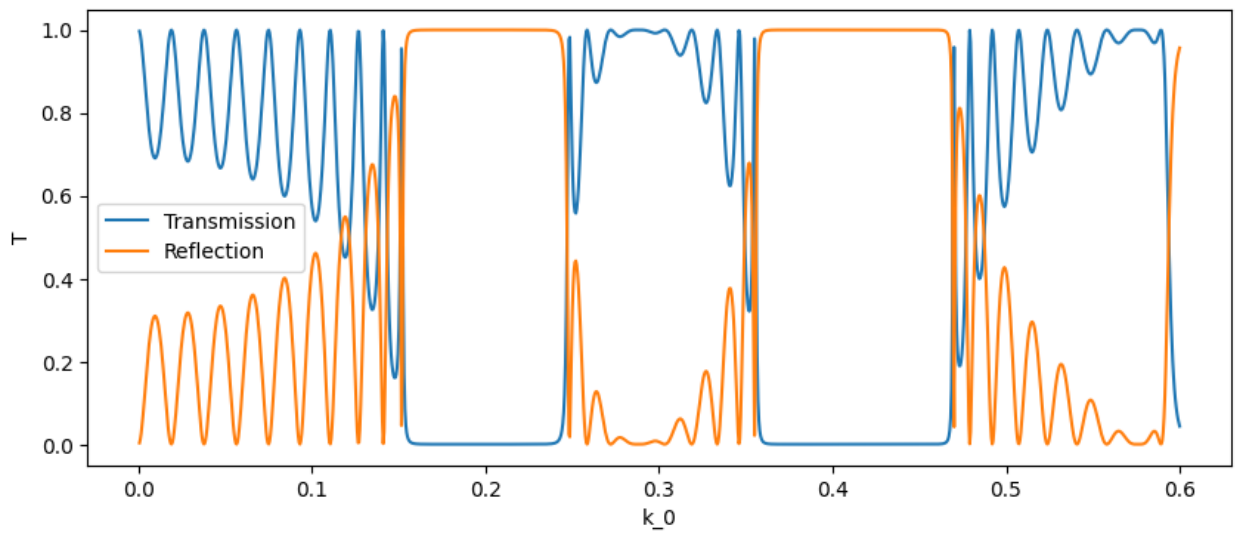
(continues on next page)

(continued from previous page)

```
for om in oml:
    st.solve(om)
    T,R = st.get_T(0,0, ordered = False), st.get_R(0,0, ordered = False)
    DATA.append((T,R))

DATA = pd.DataFrame(DATA, index = oml, columns = ['T', 'R'])
```

[4]:
```
plt.figure(figsize = (10,4))
plt.plot(DATA['T'], label = 'Transmission')
plt.plot(DATA['R'], label = 'Reflection')
plt.xlabel('k_0'), plt.ylabel('T'), plt.legend()
```

[4]:
```
(Text(0.5, 0, 'k_0'),
 Text(0, 0.5, 'T'),
 <matplotlib.legend.Legend at 0x7f45dc293b50>)
```

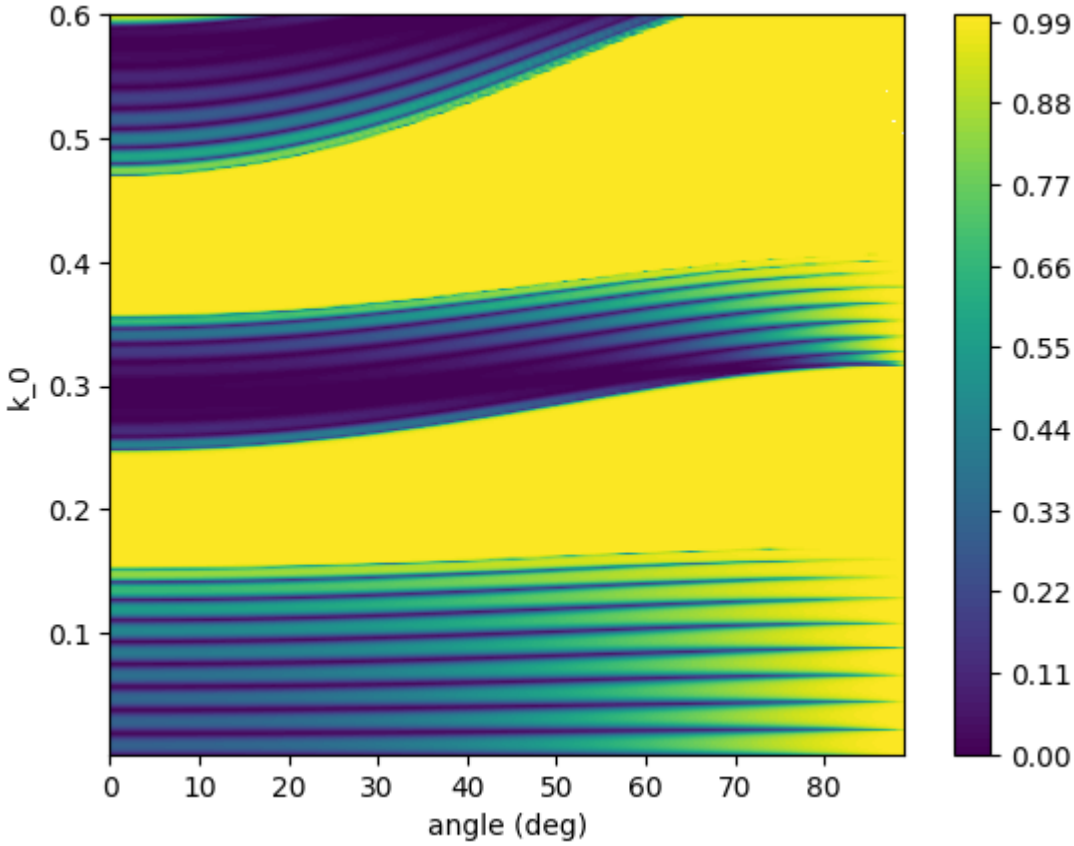


Transmission and reflection at an angle

[5]:
```
oml = np.linspace(0.002, 0.6,300)
angles = np.linspace(0,89,90)
DATA = []
for om in oml:
    for angle in angles:
        st.solve(om, kx = om*np.sqrt(2.0)*np.sin(np.pi*angle/180.0))
        RE, TE, RM, TM = st.get_R(1,1,ordered=False), st.get_T(1,1,ordered=False), st.
↪get_R(0,0,ordered=False), st.get_T(0,0,ordered=False)
        DATA.append((om, angle, RE, TE, RM, TM))

DATA = pd.DataFrame(DATA, columns = ['om', 'angle', 'R_TE', 'T_TE', 'R_TM', 'T_TM'])
```

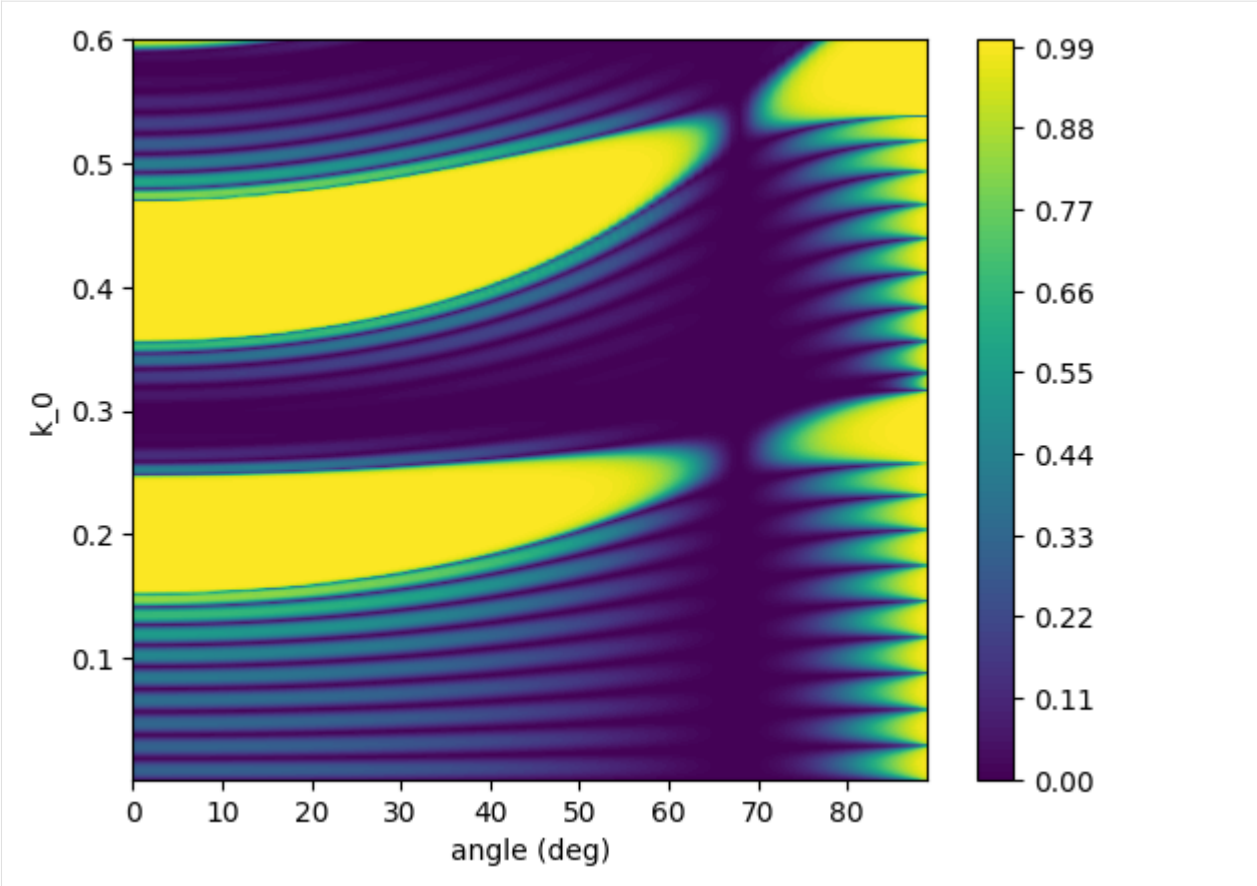
[6]:
```
OM, ANG = np.meshgrid(oml, angles, indexing = 'ij')
plt.contourf(angles, oml, DATA.pivot_table('R_TE','om', 'angle'), levels = 101)
plt.xlabel('angle (deg)'), plt.ylabel('k_0'), plt.colorbar()
```

[6]:
```
(Text(0.5, 0, 'angle (deg)'),
 Text(0, 0.5, 'k_0'),
 <matplotlib.colorbar.Colorbar at 0x7f454d1e9750>)
```



[7]:
```
plt.contourf(angles, oml, DATA.pivot_table('R_TM', 'om', 'angle'), levels = 101)
plt.xlabel('angle (deg)'), plt.ylabel('k_0'), plt.colorbar()
```

[7]:
```
(Text(0.5, 0, 'angle (deg)'),
 Text(0, 0.5, 'k_0'),
 <matplotlib.colorbar.Colorbar at 0x7f454739ffa0>)
```



Band structure infinite Bragg Grating

A convenient way to make sense of reflection and transmission spectra of a Bragg grating is to look at the band structure. This also can be done with the Aperiodic Fourier Modal Method, since it is possible, given the scattering matrix of the unit cell, to get the Bloch modes by simply solving an eigenvalue problem.

First step is to build the stack object representing the unit cell. In the following the unit cell of the same grating analyzed in the previos section. It is worth noting that, due to the way the first and last layer are handled by the code, their thickness will not be included in the unit cell. Thus, first and last layer are irrelevant to the code, but they need to be the same in order for the code to work properly.

Thus, this is how the cell is built:

[8]:
```
mat = [lay1, lay1, lay2, lay1]
d   = [0.0,0.5, 0.5,0.0]
st = A_FMM.Stack(mat, d)
eps = st.calculate_epsilon(x=[-0.5, 0.5], z = np.linspace(0.0, st.total_length, 1001))
plt.contourf(np.squeeze(eps['z']), np.squeeze(eps['x']), np.squeeze(eps['eps']), cmap=
↪'Greys')
plt.xlabel('z'), plt.ylabel('x'), plt.colorbar()
```

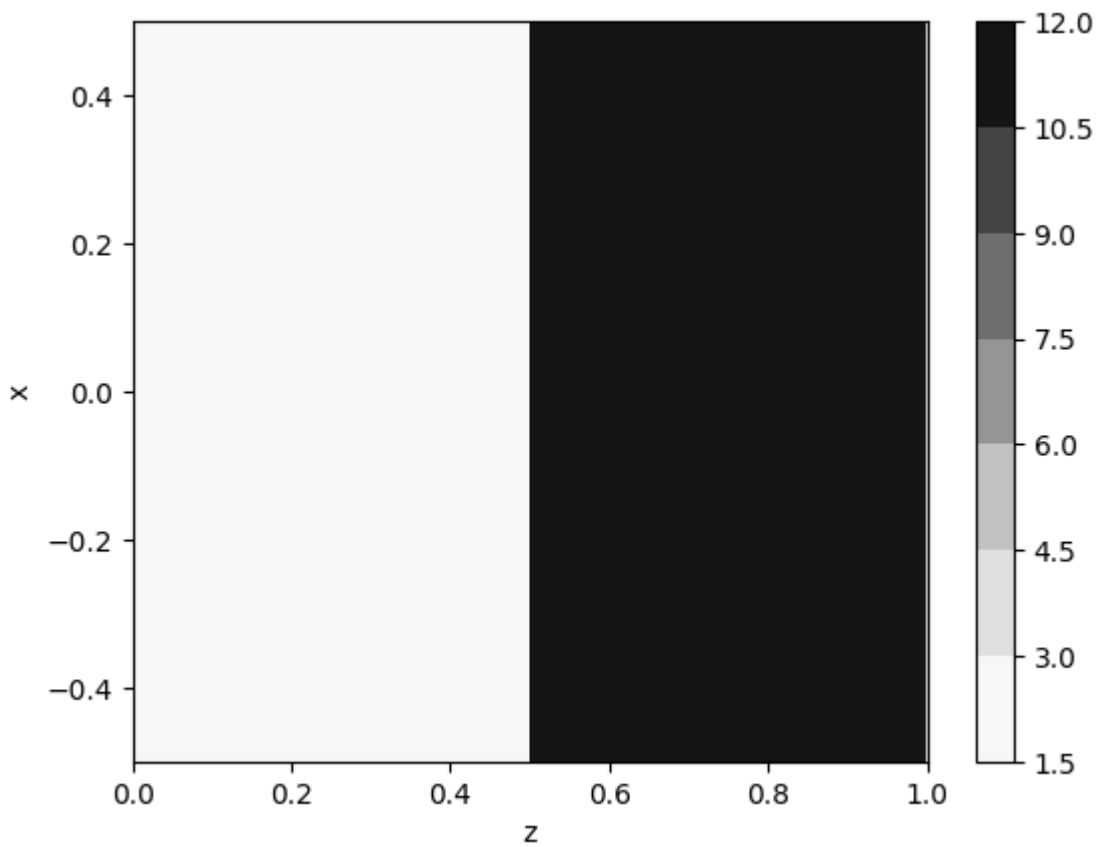
```
/home/marco/Documents/MyPrograms/A_FMM/test/lib/python3.10/site-packages/matplotlib/
↪contour.py:1568: ComplexWarning: Casting complex values to real discards the imaginary␣
↪part
```

(continues on next page)

(continued from previous page)

```
  self.zmax = z.max().astype(float)
/home/marco/Documents/MyPrograms/A_FMM/test/lib/python3.10/site-packages/matplotlib/
↪contour.py:1569: ComplexWarning: Casting complex values to real discards the imaginary␣
↪part
  self.zmin = z.min().astype(float)
/home/marco/Documents/MyPrograms/A_FMM/test/lib/python3.10/site-packages/numpy/ma/core.
↪py:2820: ComplexWarning: Casting complex values to real discards the imaginary part
  _data = np.array(data, dtype=dtype, copy=copy,
```

[8]:
```
(Text(0.5, 0, 'z'),
 Text(0, 0.5, 'x'),
 <matplotlib.colorbar.Colorbar at 0x7f454729f280>)
```



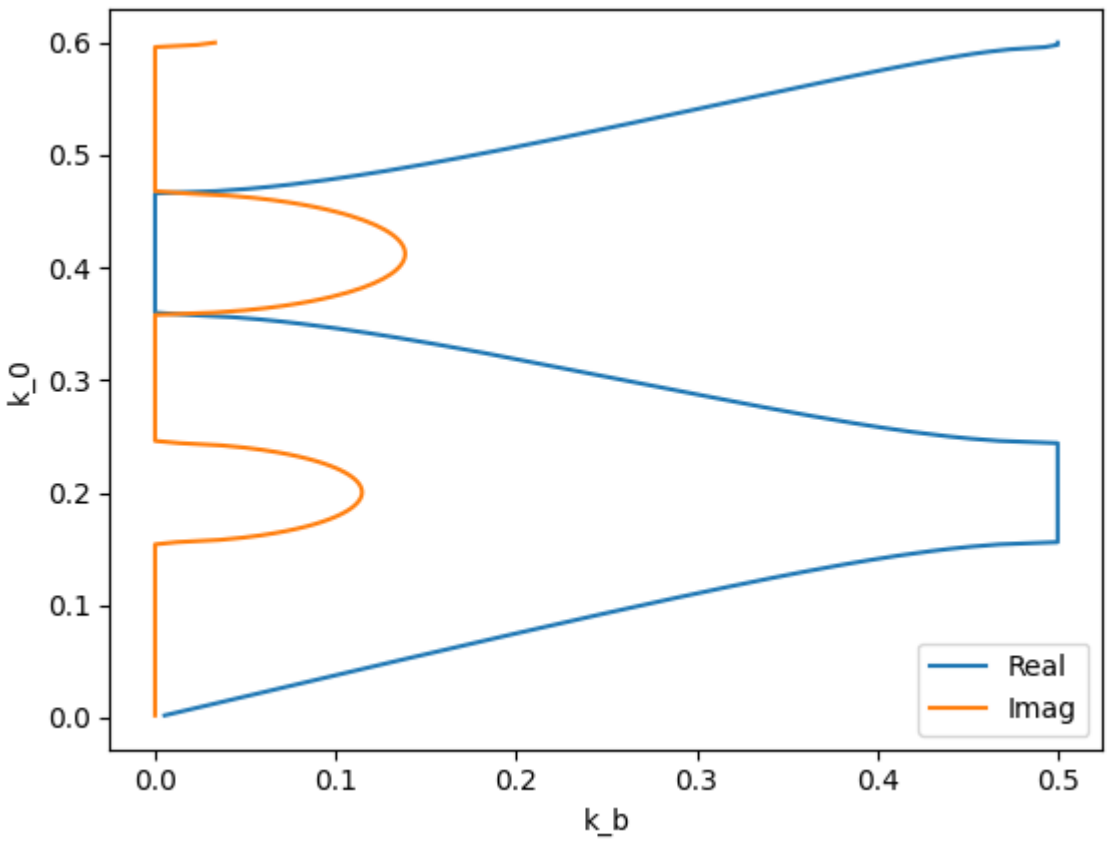
After that, getting the Bloch mode just requires to calculate the scattering matrix by calling `solve` and the solving for the Bloch vector by calling `bloch_modes`, which returns an array containing the found Bloch vectors. In general the Bloch vector is a complex quantity, where the real part represent the evolution of the phase and the imaginary part the evolution of the amplitude. The returned Bloch vectors are ordered by increasing imaginary part.

[9]:
```
DATA = []
for om in oml:
    st.solve(om)
    bm = st.bloch_modes()
    DATA.append([om, bm[0]])

DATA = pd.DataFrame(DATA, columns = ['om', 'bk'])
```

The obtained Bloch vector can then be plotted against the frequency:

[10]:
```
plt.plot(np.abs(np.real(DATA['bk'])), DATA['om'], label='Real')
plt.plot(np.imag(DATA['bk']), DATA['om'], label='Imag')
plt.xlabel('k_b'), plt.ylabel('k_0'), plt.legend()
```

[10]:
```
(Text(0.5, 0, 'k_b'),
 Text(0, 0.5, 'k_0'),
 <matplotlib.legend.Legend at 0x7f4547566d70>)
```



2.3 Full Calculation

2.3.1 PML Test (Hugonin 2005)

This test illustrates the usage of the complex coordinate transform. This is useful when dealing with structure in which a high portion of the power is scattered away.

This test is based on the paper **"Perfectly matched layers as nonlinear coordinate transforms: a generalized formalization" by Jean Paul Hugonin and Philippe Lalanne (J. Opt. Soc. Am. A / Vol. 22, No. 9 / September 2005).** 10.1364/JOSAA.22.001844

Summary

The example will compute the reflection and transmission from 2 dent in a 1D waveguide.

The following example will: 1. Import all necessary modules 2. Define the layers involved 3. Combine them into a structure 4. Calculate transmission and reflection 5. Sweep over truncation order

Import modules

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import A_FMM
```

Define layers involved

[2]:
```
ax = 1.1
lam = 0.975
k0 = ax/lam

s = 0.3
d = 0.15

n_core = 3.5
n_clad = 2.9
n_air = 1.0

Nx = 20
Ny = 0

cr = A_FMM.Creator()
cr.slab(n_core**2.0, n_clad**2.0, n_air**2.0, s/ax)
wave = A_FMM.Layer(Nx,0,cr)
wave.transform(0.3/1.1, complex_transform=True)
cr.slab(n_air**2.0, n_clad**2.0, n_air**2.0, s/ax)
gap=A_FMM.Layer(Nx,0,cr)
gap.transform(0.3/1.1, complex_transform=True)
```

[2]:
```
(array([[ 5.90909091e-01-4.54545455e-02j,  2.72660178e-01+1.73262321e-02j,
         -4.96979855e-02+2.09010219e-02j, ...,
         -7.50191184e-06-2.50721126e-06j, -6.93858730e-06-2.31863380e-06j,
         -1.99160242e-06-6.65441959e-07j],
        [ 2.72660178e-01+1.73262321e-02j,  5.90909091e-01-4.54545455e-02j,
          2.72660178e-01+1.73262321e-02j, ...,
         -2.51739592e-06-8.41459341e-07j, -7.50191184e-06-2.50721126e-06j,
         -6.93858730e-06-2.31863380e-06j],
        [-4.96979855e-02+2.09010219e-02j,  2.72660178e-01+1.73262321e-02j,
          5.90909091e-01-4.54545455e-02j, ...,
          5.24552476e-06+1.75363218e-06j, -2.51739592e-06-8.41459341e-07j,
         -7.50191184e-06-2.50721126e-06j],
        ...,
```

(continues on next page)

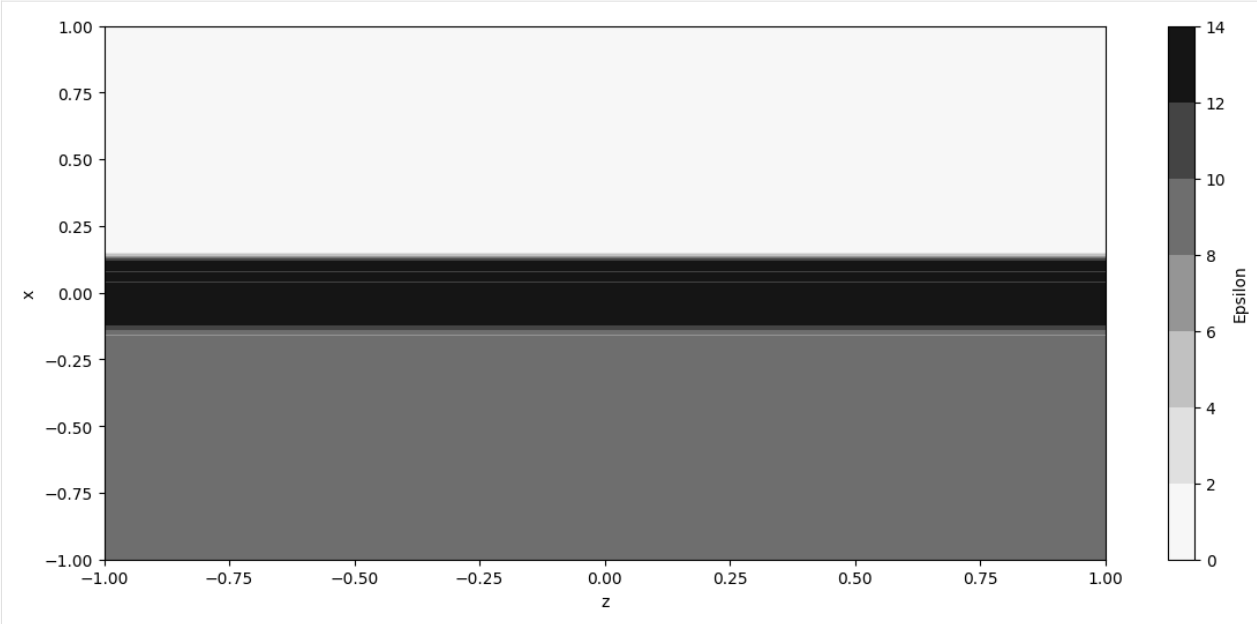
(continued from previous page)

```
        [-7.50191184e-06-2.50721126e-06j, -2.51739592e-06-8.41459341e-07j,
          5.24552476e-06+1.75363218e-06j, ...,
          5.90909091e-01-4.54545455e-02j,  2.72660178e-01+1.73262321e-02j,
         -4.96979855e-02+2.09010219e-02j],
        [-6.93858730e-06-2.31863380e-06j, -7.50191184e-06-2.50721126e-06j,
         -2.51739592e-06-8.41459341e-07j, ...,
          2.72660178e-01+1.73262321e-02j,  5.90909091e-01-4.54545455e-02j,
          2.72660178e-01+1.73262321e-02j],
        [-1.99160242e-06-6.65441959e-07j, -6.93858730e-06-2.31863380e-06j,
         -7.50191184e-06-2.50721126e-06j, ...,
         -4.96979855e-02+2.09010219e-02j,  2.72660178e-01+1.73262321e-02j,
          5.90909091e-01-4.54545455e-02j]]),
 None)
```

[3]:
```
x = np.linspace(-1.0, 1.0, 101)
z = np.linspace(-1.0, 1.0, 101)

eps = wave.calculate_epsilon(x=x, z=z)
fig, axp = plt.subplots(1,1, figsize=(14, 6))
_ = axp.contourf(np.squeeze(z), x, np.squeeze(eps['eps']), cmap='Greys')
plt.xlabel('z'), plt.ylabel('x'), fig.colorbar(_, ax=axp).set_label('Epsilon')
```

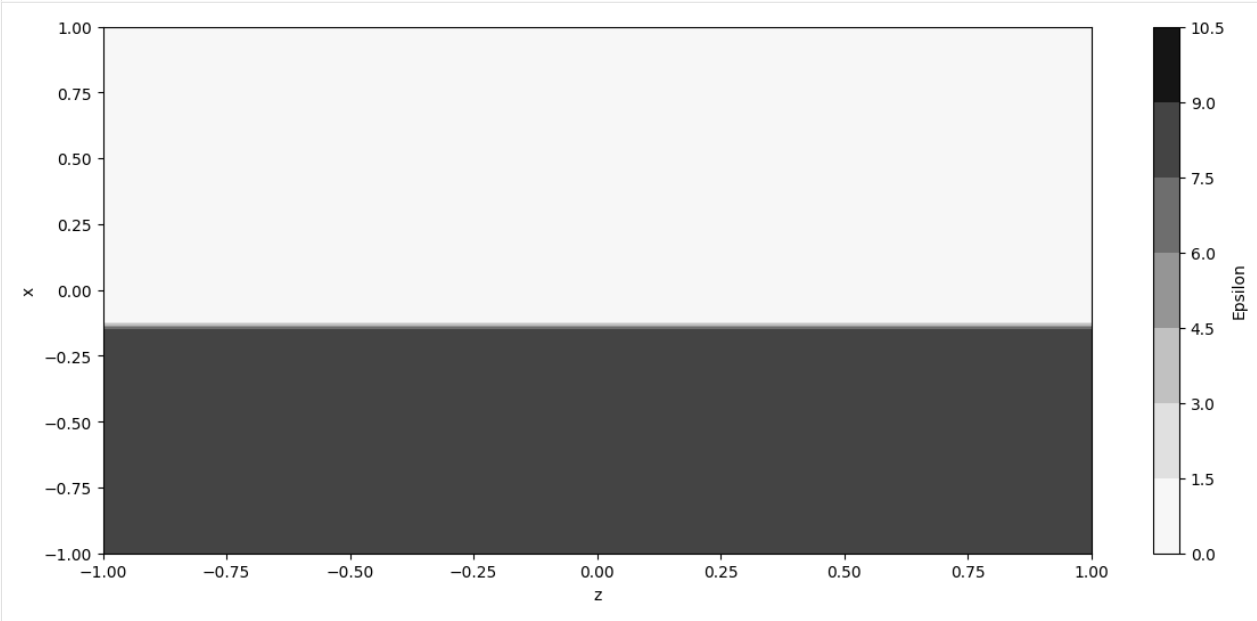
```
/home/marco/Documents/MyPrograms/A_FMM/test_venv/lib/python3.10/site-packages/matplotlib/
↪contour.py:1568: ComplexWarning: Casting complex values to real discards the imaginary␣
↪part
  self.zmax = z.max().astype(float)
/home/marco/Documents/MyPrograms/A_FMM/test_venv/lib/python3.10/site-packages/matplotlib/
↪contour.py:1569: ComplexWarning: Casting complex values to real discards the imaginary␣
↪part
  self.zmin = z.min().astype(float)
/home/marco/Documents/MyPrograms/A_FMM/test_venv/lib/python3.10/site-packages/numpy/ma/
↪core.py:2820: ComplexWarning: Casting complex values to real discards the imaginary␣
↪part
  _data = np.array(data, dtype=dtype, copy=copy,
```

[3]:
```
(Text(0.5, 0, 'z'), Text(0, 0.5, 'x'), None)
```



[4]:
```
eps = gap.calculate_epsilon(x=x, z=z)
fig, axp = plt.subplots(1,1, figsize=(14, 6))
_ = axp.contourf(z, x, np.squeeze(eps['eps']), cmap='Greys')
plt.xlabel('z'), plt.ylabel('x'), fig.colorbar(_, ax=axp).set_label('Epsilon')
```

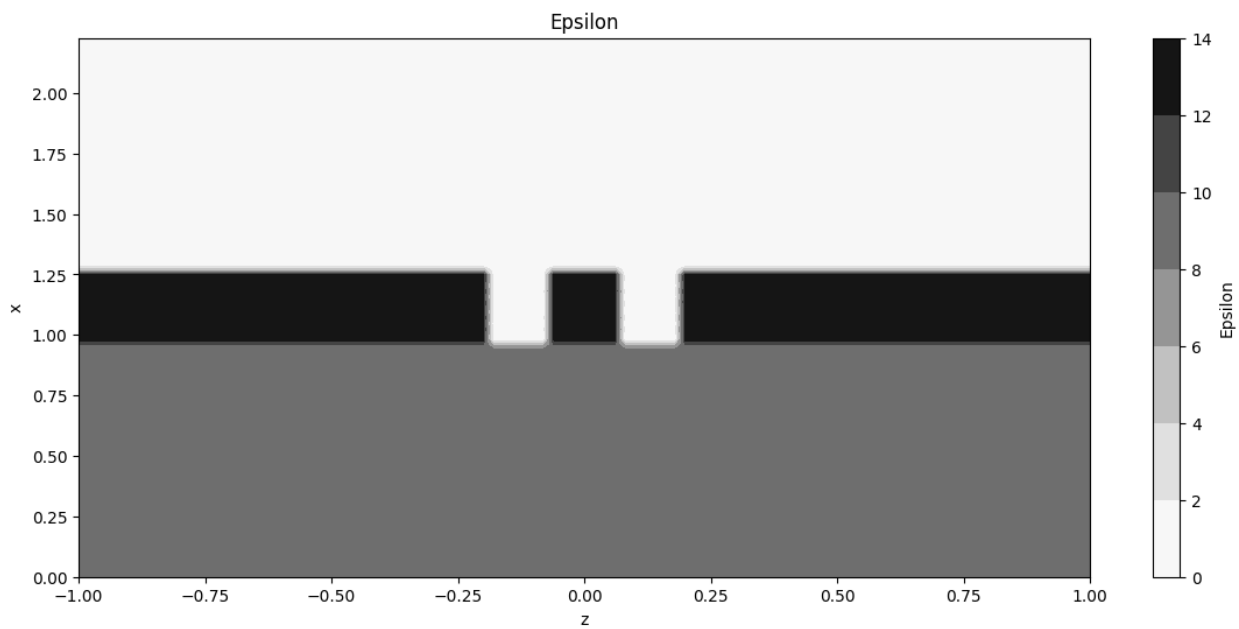
[4]:
```
(Text(0.5, 0, 'z'), Text(0, 0.5, 'x'), None)
```



Define stack structure

[5]:
```
mat = [wave, gap, wave, gap, wave]
dl = [_/ax for _ in [1.0,d,d,d,1.0]]
st = A_FMM.Stack(mat, dl)
st.count_interface()
st.transform(0.3/1.1, complex_transform=True)

x = np.linspace(-1.0, 1.0, 101)
z = np.linspace(0.0, st.total_length, 101)

fig, axp = plt.subplots(1,1, figsize=(14,6))
eps = st.calculate_epsilon(x=x, z=z)
_ = axp.contourf(x, z, np.squeeze(eps['eps']), cmap='Greys')
axp.set_xlabel('z'), axp.set_ylabel('x'),  fig.colorbar(_, ax=axp).set_label('Epsilon'),␣
↪axp.set_title('Epsilon')
```

[5]:
```
(Text(0.5, 0, 'z'), Text(0, 0.5, 'x'), None, Text(0.5, 1.0, 'Epsilon'))
```



Solve structure and calculate reflection

[6]:
```
st.solve(ax/lam)
print('TE Reflection:{}'.format(st.get_R(0,0)))
print('TM Reflection:{}'.format(st.get_R(1,1)))
```

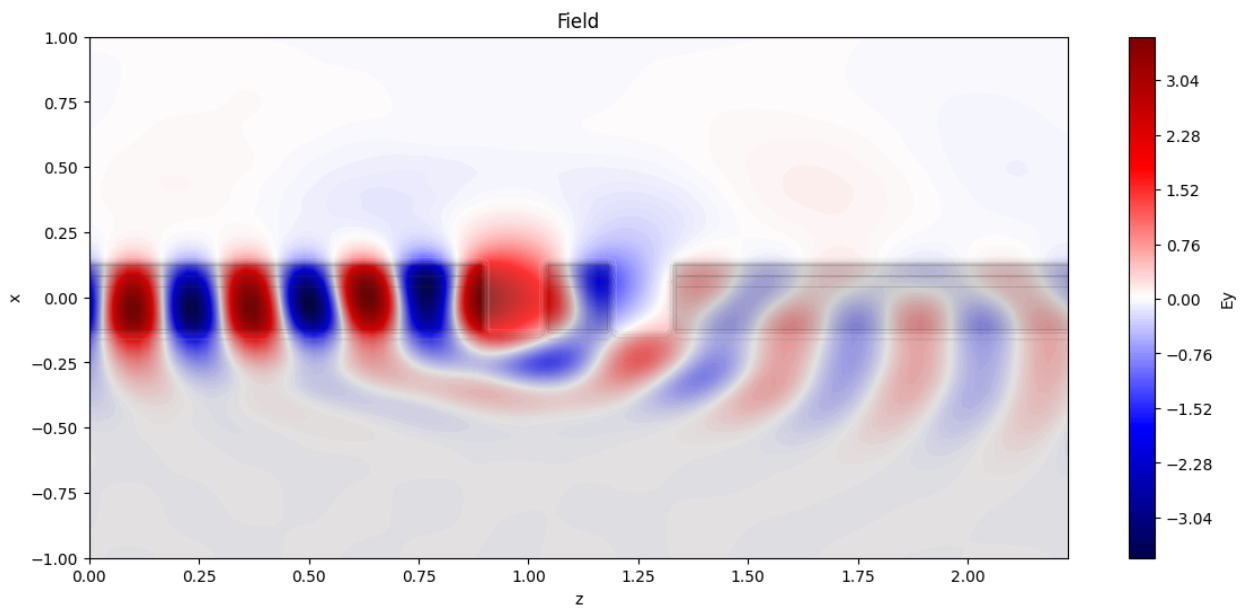
```
TE Reflection:0.39516541327755816
TM Reflection:0.3562340123876459
```

Field Plotting

Plotting field under TE illumination

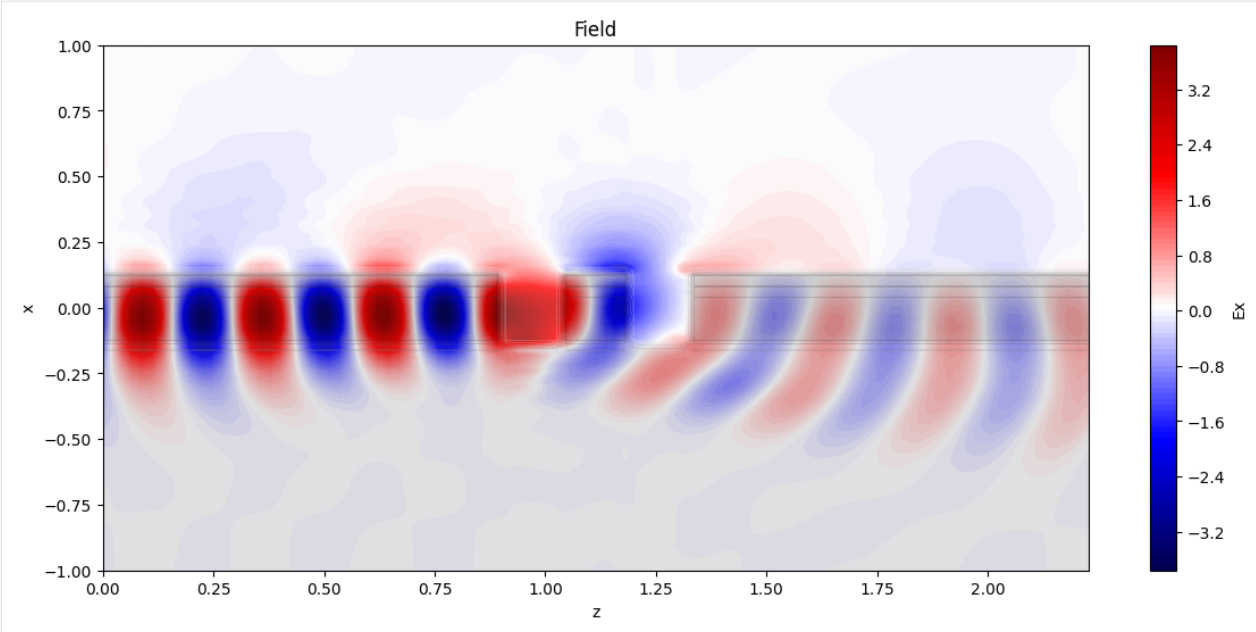
[7]:
```
u = wave.create_input({0 : 1.0})
field = st.calculate_fields(u1=u, x=x, z=z)
fig, axp = plt.subplots(1,1, figsize=(14,6))
_ = axp.contourf(z, x, np.squeeze(field['Ey']), cmap='seismic', levels=201)
axp.contourf(z, x, np.squeeze(eps['eps']), cmap='Greys', alpha=0.2)
axp.set_xlabel('z'), axp.set_ylabel('x'),  fig.colorbar(_, ax=axp).set_label('Ey'), axp.
↪set_title('Field')
```

[7]:
```
(Text(0.5, 0, 'z'), Text(0, 0.5, 'x'), None, Text(0.5, 1.0, 'Field'))
```



[8]:
```
u = wave.create_input({1 : 1.0})
field = st.calculate_fields(u1=u, x=x, z=z)
fig, axp = plt.subplots(1,1, figsize=(14,6))
_ = axp.contourf(z, x, np.squeeze(field['Ex']), cmap='seismic', levels=101)
axp.contourf(z, x, np.squeeze(eps['eps']), cmap='Greys', alpha=0.2)
axp.set_xlabel('z'), axp.set_ylabel('x'),  fig.colorbar(_, ax=axp).set_label('Ex'), axp.
↪set_title('Field')
```

[8]:
```
(Text(0.5, 0, 'z'), Text(0, 0.5, 'x'), None, Text(0.5, 1.0, 'Field'))
```



Sweep over truncation order

[9]:
```
def calc(Nx):
    cr.slab(n_core**2.0, n_clad**2.0, n_air**2.0, s/ax)
    wave = A_FMM.Layer(Nx,0,cr)
    cr.slab(n_air**2.0, n_clad**2.0, n_air**2.0, s/ax)
    gap=A_FMM.Layer(Nx,0,cr)
    mat = [wave, gap, wave, gap, wave]
    dl = [x/ax for x in [1.0,d,d,d,1.0]]
    st = A_FMM.Stack(mat, dl)
    st.count_interface()
    st.transform(0.7, complex_transform=True)
    st.solve(ax/lam)
    return st.get_R(0,0), st.get_R(1,1)
```

[10]:
```
NX = [10,20,30,40,50,60,70,80,100, 110, 120, 130, 140, 150, 160, 170, 180, 190, 200, 210,
↪ 220, 230, 240]
RR = [calc(Nx) for Nx in NX]
Data = pd.DataFrame(RR, index=NX, columns=['TE', 'TM'])
```

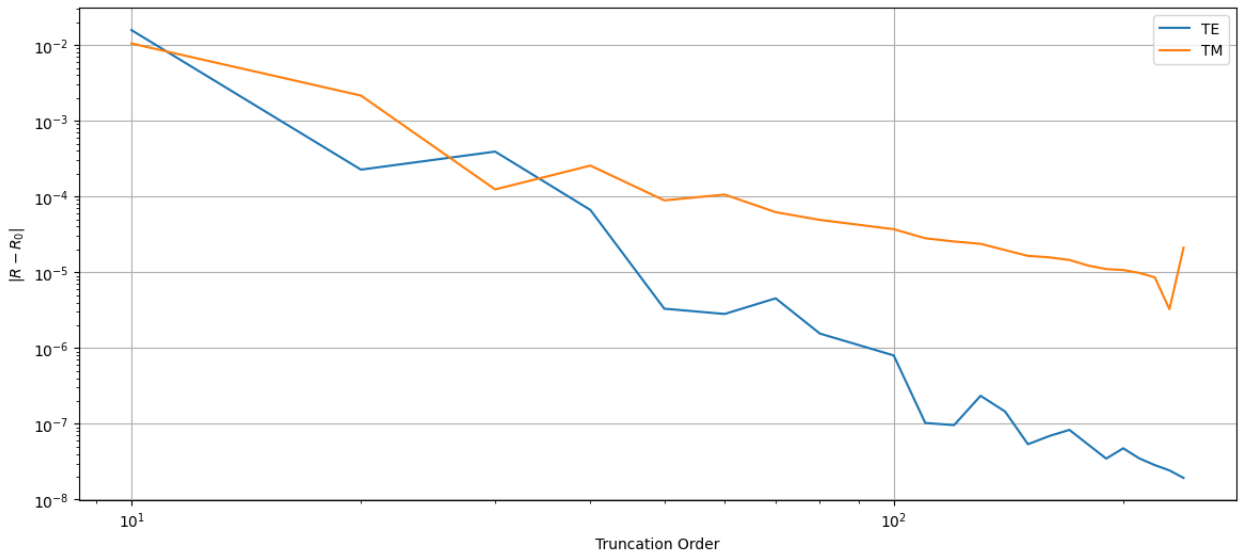
[11]:
```
fig, axp = plt.subplots(1,1, figsize=(14, 6))
plt.plot(NX, abs(Data['TE']-0.3952113445) , label='TE')
plt.plot(NX, abs(Data['TM']-0.3554787) , label='TM')
plt.yscale('log'), plt.xscale('log'), plt.xlabel('Truncation Order'), plt.ylabel(r'$|R-R_
↪0|$'), plt.legend(), plt.grid()
```

[11]:
```
(None,
 None,
 Text(0.5, 0, 'Truncation Order'),
 Text(0, 0.5, '$|R-R_0|$'),
```

(continues on next page)

(continued from previous page)

```
<matplotlib.legend.Legend at 0x7f5bd2d3c2e0>,
None)
```



2.3.2 2D Grating

This example illustrates the use of A-FMM to calculate reflection and transmission from a 2D grating.

The calculations shown here are based on the paper: **Lee, Sun-Goo, et al. "Polarization-independent electromagnetically induced transparency-like transmission in coupled guided-mode resonance structures." Applied Physics Letters 110.11 (2017): 111106.** 10.1063/1.4978670

Importing modules

[1]:
```
import numpy as np
import matplotlib.pyplot as plt

import A_FMM

# general parameter of calculation
N = 5  # truncation order for Fourier expansion
```

Definition of layers

The structure under investigation is a 2D grating of square dielectric (n=2) pillars on top o two slabs of the same dielectric, all embedded in ait (n=1). The grating is build on a square lattice with lattice constant **a=1**.

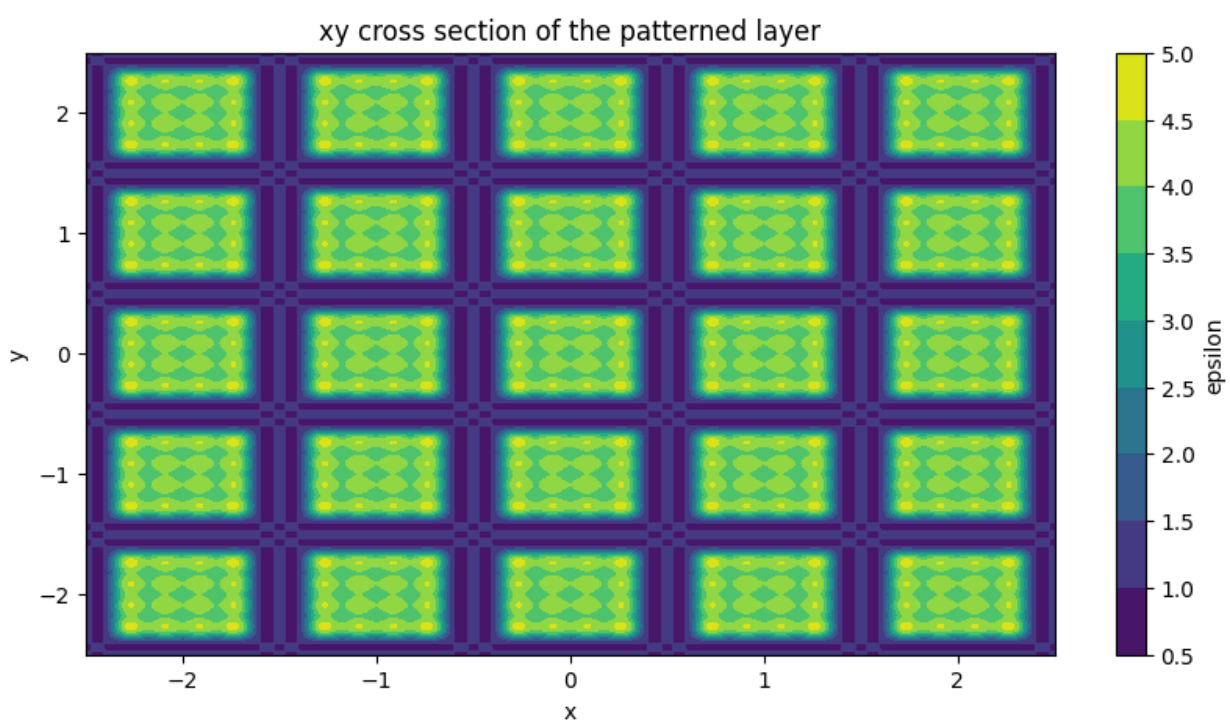
The following code defines the needed layers.

Patterned layer

This layer is built in the usual way, by defining a Creator object and call one of the pre-defined geometries.

[2]:
```
# definition of parameter of layer
w_pc = 0.7

# define the creator and layer
creator = A_FMM.Creator()
creator.rect(eps_core=4.0, eps_clad=1.0, w=w_pc, h=w_pc)
pattern = A_FMM.Layer(Nx=N, Ny=N, creator=creator)

# plotting dielectric constant profile of the layer
x = np.linspace(-2.5, 2.5, 501)
y = np.linspace(-2.5, 2.5, 501)

eps = pattern.calculate_epsilon(x=x, y=y)

fig, ax = plt.subplots(1,1, figsize = (10,5))
contour = ax.contourf(np.squeeze(eps['x']), np.squeeze(eps['y']), np.squeeze(eps['eps'].
↪real))
fig.colorbar(contour, ax=ax, label='epsilon')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_title('xy cross section of the patterned layer')
```

[2]:
```
Text(0.5, 1.0, 'xy cross section of the patterned layer')
```



Uniform layers

Since these layer have no patterning they are defined by calling Layer_uniform.

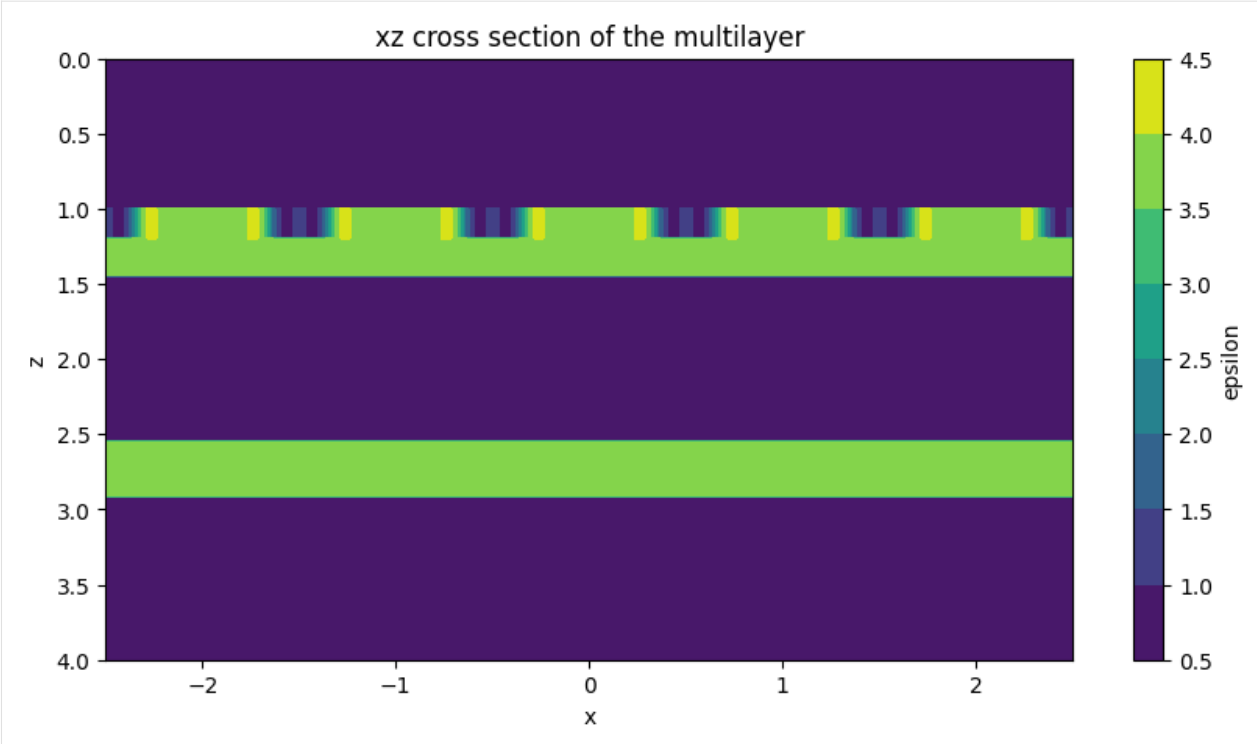
[3]:
```
# define uniform layers
background = A_FMM.Layer_uniform(Nx=N, Ny=N, eps=1.0)
dielectric = A_FMM.Layer_uniform(Nx=N, Ny=N, eps=4.0)
```

Stack definition

The following code defines the 3D stack.

[4]:
```
# definition of relevant parameters for the stack (check the paper for parameters␣
↪meaning)
t_pc = 0.2
t_1 = 0.25
t_2 = 0.361
h = 1.1

# definition of material and thicknesses lists
layers = [background, pattern, dielectric, background, dielectric, background]
thicknesses = [1.0, t_pc, t_1, h, t_2, 1.0]

# stack definition
stack = A_FMM.Stack(layers=layers, d=thicknesses)

# plotting dielectric constant profile of a cross section of the stack
z = np.linspace(-1.0, 4.0, 501)
eps = stack.calculate_epsilon(x=x, z=z)

fig, ax = plt.subplots(1,1, figsize = (10,5))
contour = ax.contourf(np.squeeze(eps['x']), np.squeeze(eps['z']), np.squeeze(eps['eps'].
↪real))
ax.invert_yaxis()
fig.colorbar(contour, ax=ax, label='epsilon')
ax.set_xlabel('x')
ax.set_ylabel('z')
ax.set_title('xz cross section of the multilayer')
```

[4]:
```
Text(0.5, 1.0, 'xz cross section of the multilayer')
```

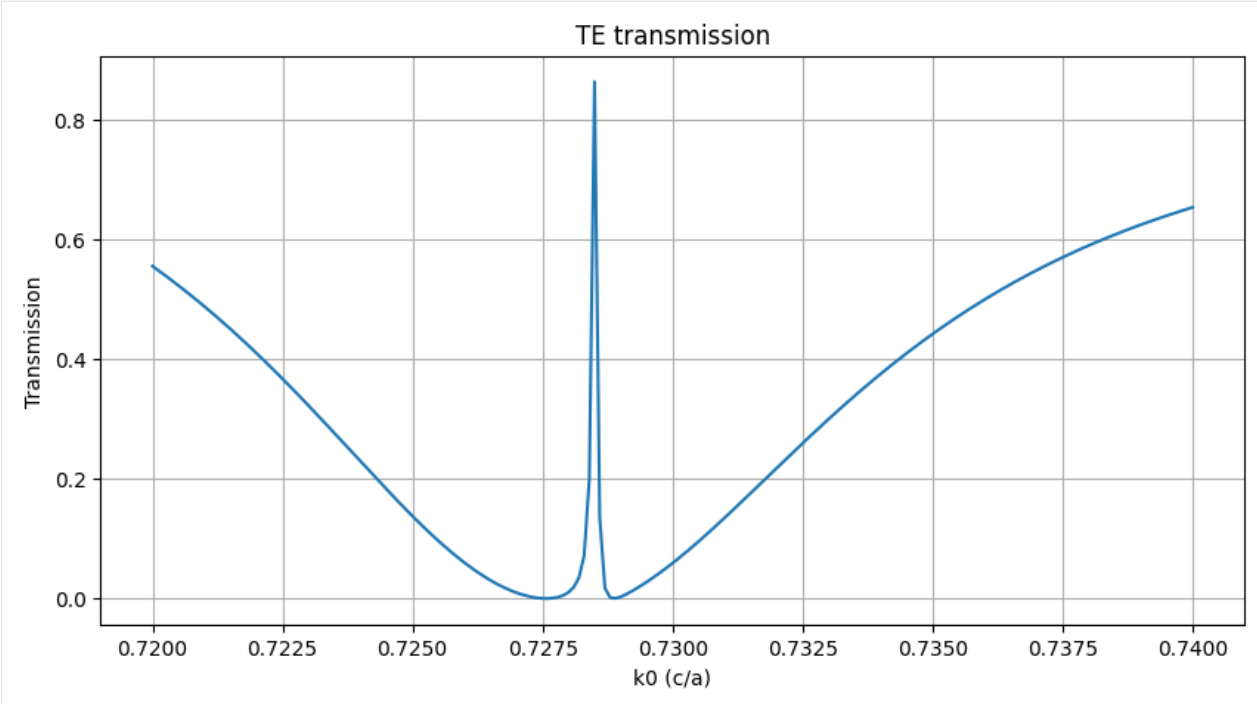


Transmission calculation

The following code calculates the transmission.

[5]:
```
# definition of the array of frequency to be used for the calculation
k0_array = np.linspace(0.72, 0.74, 201)
transmission = []

# loop over the frequencies for calculating transmission
for k0 in k0_array:
    stack.solve(k0=k0)
    transmission.append(stack.get_T(stack.NPW//2, stack.NPW//2, ordered=False))

fig, ax = plt.subplots(1,1, figsize = (10,5))
ax.plot(k0_array, transmission)
ax.set_xlabel('k0 (c/a)')
ax.set_ylabel('Transmission')
ax.grid()
ax.set_title('TE transmission')
```

[5]:
```
Text(0.5, 1.0, 'TE transmission')
```



Note: get_T mode ordering is designed to work for guided modes, so the modes are ordered by decreasing effective index. When the layer involved are uniform and the eigenmodes are plane waves, ordering the modes is not the best strategy, especially for not normal incidence. For uniform layers solved with a number **N** of fourier components, one obtains **2N** plane waves. The solutions are divided into to groups: the first **N** have polarization along x, the other have polarization along y. The order 0 sits in the middle of each group. The easy way to obtain the correct mode is to use **layer.NPW//2** if the polarization is along x and **layer.NPW//2 + layer.NPW** if the polarization is along y.

If other orders are needed, check **layer.G**. It is a dictionary linking the modes index with the tuple representing the x and y order.

[6]:
```
print(pattern.G)
print(pattern.NPW//2)
```

```
{0: (-5, -5), 1: (-5, -4), 2: (-5, -3), 3: (-5, -2), 4: (-5, -1), 5: (-5, 0), 6: (-5, 1),
↪ 7: (-5, 2), 8: (-5, 3), 9: (-5, 4), 10: (-5, 5), 11: (-4, -5), 12: (-4, -4), 13: (-4,␣
↪-3), 14: (-4, -2), 15: (-4, -1), 16: (-4, 0), 17: (-4, 1), 18: (-4, 2), 19: (-4, 3),␣
↪20: (-4, 4), 21: (-4, 5), 22: (-3, -5), 23: (-3, -4), 24: (-3, -3), 25: (-3, -2), 26:␣
↪(-3, -1), 27: (-3, 0), 28: (-3, 1), 29: (-3, 2), 30: (-3, 3), 31: (-3, 4), 32: (-3, 5),
↪ 33: (-2, -5), 34: (-2, -4), 35: (-2, -3), 36: (-2, -2), 37: (-2, -1), 38: (-2, 0), 39:
↪ (-2, 1), 40: (-2, 2), 41: (-2, 3), 42: (-2, 4), 43: (-2, 5), 44: (-1, -5), 45: (-1, -
↪4), 46: (-1, -3), 47: (-1, -2), 48: (-1, -1), 49: (-1, 0), 50: (-1, 1), 51: (-1, 2),␣
↪52: (-1, 3), 53: (-1, 4), 54: (-1, 5), 55: (0, -5), 56: (0, -4), 57: (0, -3), 58: (0, -
↪2), 59: (0, -1), 60: (0, 0), 61: (0, 1), 62: (0, 2), 63: (0, 3), 64: (0, 4), 65: (0,␣
↪5), 66: (1, -5), 67: (1, -4), 68: (1, -3), 69: (1, -2), 70: (1, -1), 71: (1, 0), 72:␣
↪(1, 1), 73: (1, 2), 74: (1, 3), 75: (1, 4), 76: (1, 5), 77: (2, -5), 78: (2, -4), 79:␣
↪(2, -3), 80: (2, -2), 81: (2, -1), 82: (2, 0), 83: (2, 1), 84: (2, 2), 85: (2, 3), 86:␣
↪(2, 4), 87: (2, 5), 88: (3, -5), 89: (3, -4), 90: (3, -3), 91: (3, -2), 92: (3, -1),␣
↪93: (3, 0), 94: (3, 1), 95: (3, 2), 96: (3, 3), 97: (3, 4), 98: (3, 5), 99: (4, -5),␣
↪100: (4, -4), 101: (4, -3), 102: (4, -2), 103: (4, -1), 104: (4, 0), 105: (4, 1), 106:␣
↪(4, 2), 107: (4, 3), 108: (4, 4), 109: (4, 5), 110: (5, -5), 111: (5, -4), 112: (5, -
↪3), 113: (5, -2), 114: (5, -1), 115: (5, 0), 116: (5, 1), 117: (5, 2), 118: (5, 3),␣
```

(continues on next page)

(continued from previous page)

```
↪119: (5, 4), 120: (5, 5)}
60
```

2.3.3 Layer from function and PhC Slab

This example illustrate the use of a function to define the dieletric profile of the unit cell. This is then used to calculate reflection and transmission trhough a Photonic Crystal Slab with circular holes.

The results presented here are respoduciton from the paper: **Fan, Shanhui, and John D. Joannopoulos. "Analysis of guided resonances in photonic crystal slabs." Physical Review B 65.23 (2002): 235112** 10.1103/PhysRevB.65.235112.

Import packages

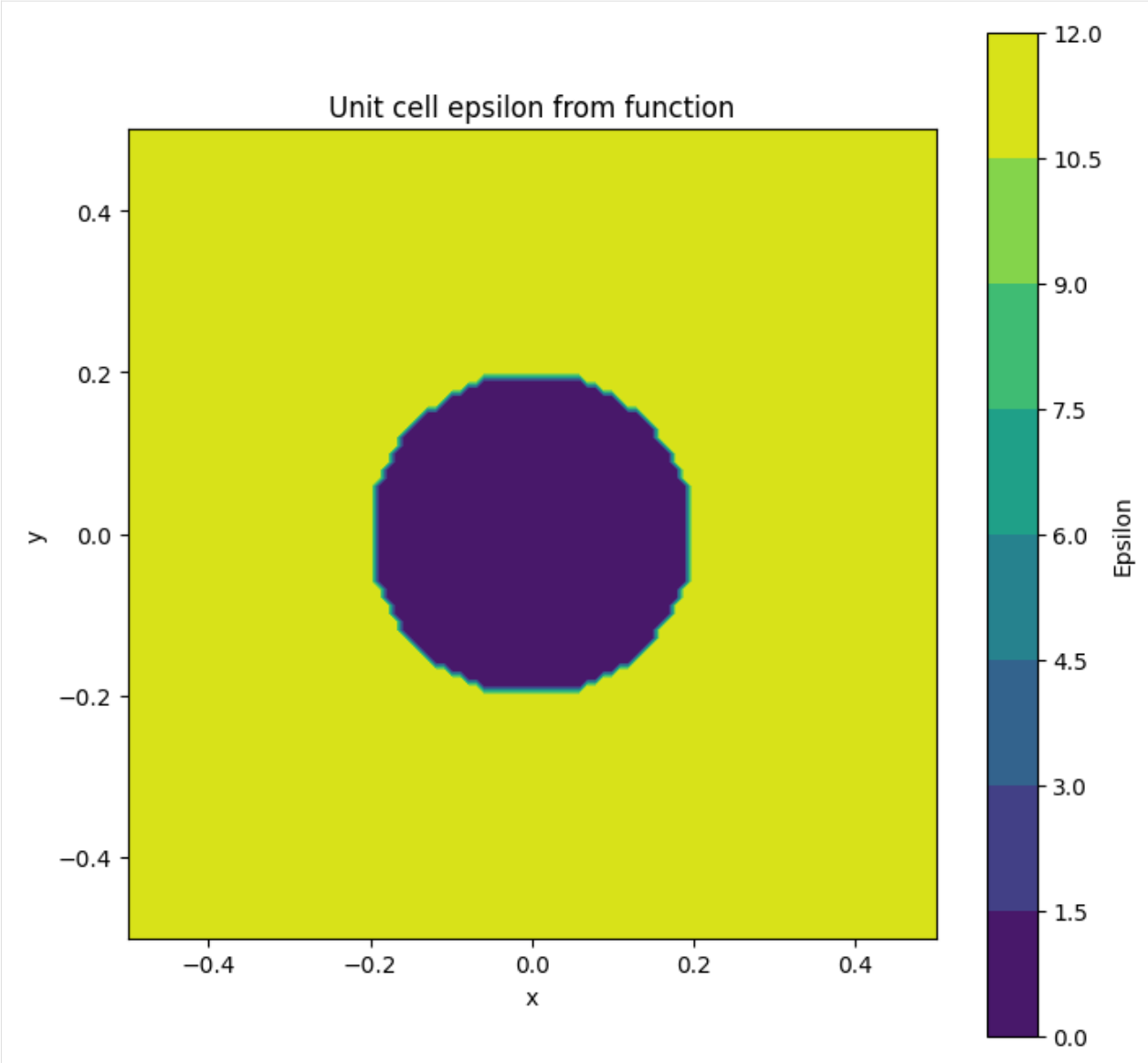
[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import A_FMM
```

Structure definition

Definition of dielectric function

[2]:
```
@np.vectorize
def eps_func(x,y, radius):
    if np.sqrt(x**2.0 + y**2.0) < radius:
        return 1.0
    return 12.0

X, Y = np.meshgrid(np.linspace(-0.5, 0.5, 101), np.linspace(-0.5, 0.5, 101), indexing='ij
↪')
EPS = eps_func(X,Y, 0.2)
fig, ax = plt.subplots(1,1, figsize = (8,8))
_ = ax.contourf(X,Y,EPS)
ax.set_xlabel('x'), ax.set_ylabel('y'), ax.set_aspect('equal', 'box'), ax.set_title(
↪'Unit cell epsilon from function')
plt.colorbar(_, ax=ax, label='Epsilon')
```

[2]:
```
<matplotlib.colorbar.Colorbar at 0x7fd374f58970>
```



Definition of patterned layer

[3]:
```
N = 5
slab = A_FMM.Layer_num(N, N, eps_func, args = (0.2,))
_ = np.linspace(-1.5, 1.5, 301)
eps = slab.calculate_epsilon(x=_, y=_)
fig, ax = plt.subplots(1,1, figsize = (8,8))
_ = ax.contourf(np.squeeze(eps['x']), np.squeeze(eps['y']), np.squeeze(eps['eps']))
ax.set_xlabel('x'), ax.set_ylabel('y'), ax.set_aspect('equal', 'box'), ax.set_title(
↪'Epsilon from fourier expansion')
plt.colorbar(_, ax=ax, label='Epsilon')
```

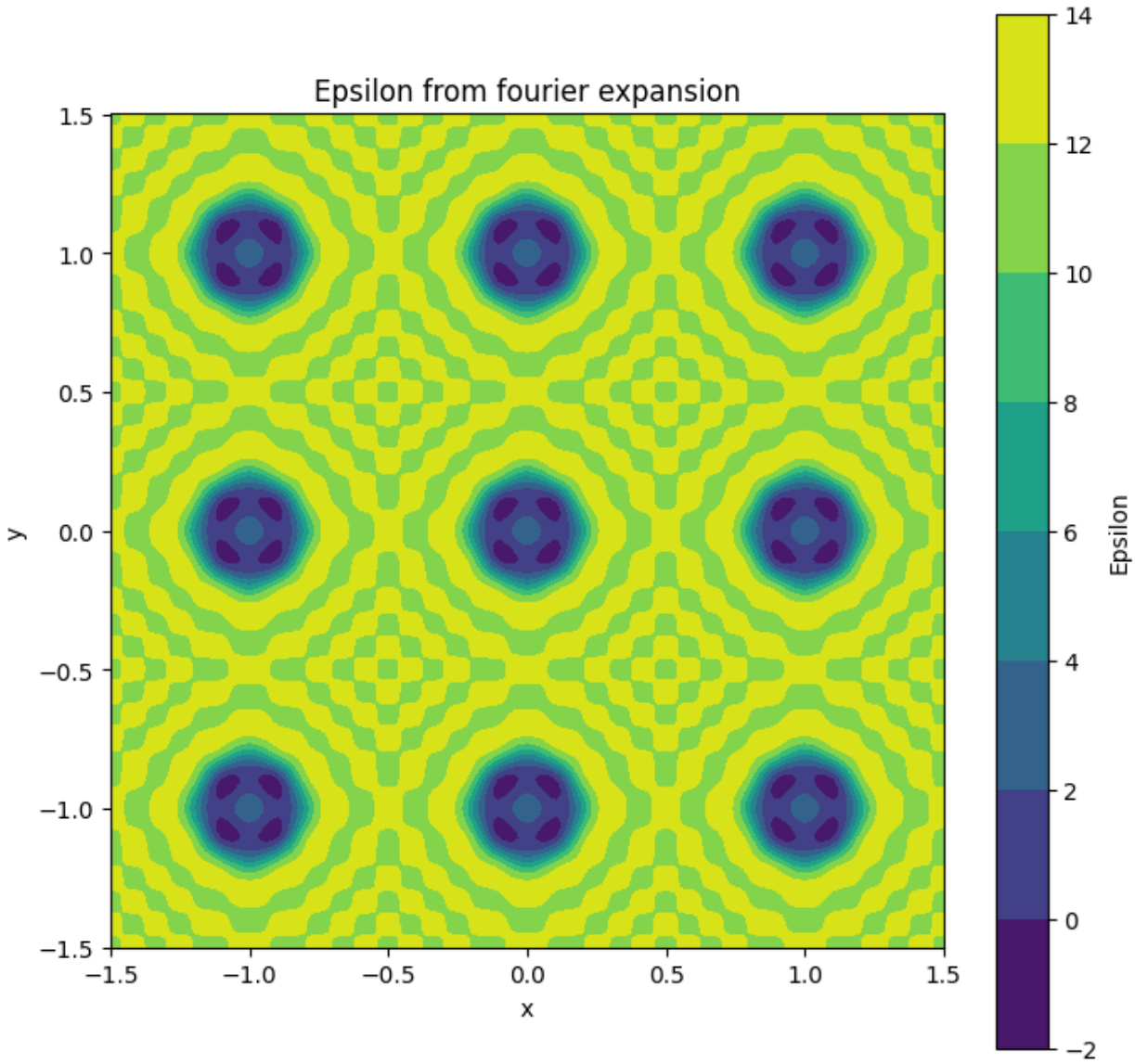
```
/home/marco/Documents/MyPrograms/A_FMM/test_venv/lib/python3.10/site-packages/matplotlib/
```

(continues on next page)

(continued from previous page)

```
↪contour.py:1568: ComplexWarning: Casting complex values to real discards the imaginary␣
↪part
  self.zmax = z.max().astype(float)
/home/marco/Documents/MyPrograms/A_FMM/test_venv/lib/python3.10/site-packages/matplotlib/
↪contour.py:1569: ComplexWarning: Casting complex values to real discards the imaginary␣
↪part
  self.zmin = z.min().astype(float)
/home/marco/Documents/MyPrograms/A_FMM/test_venv/lib/python3.10/site-packages/numpy/ma/
↪core.py:2820: ComplexWarning: Casting complex values to real discards the imaginary␣
↪part
  _data = np.array(data, dtype=dtype, copy=copy,
```

[3]:
```
<matplotlib.colorbar.Colorbar at 0x7fd360b0e950>
```



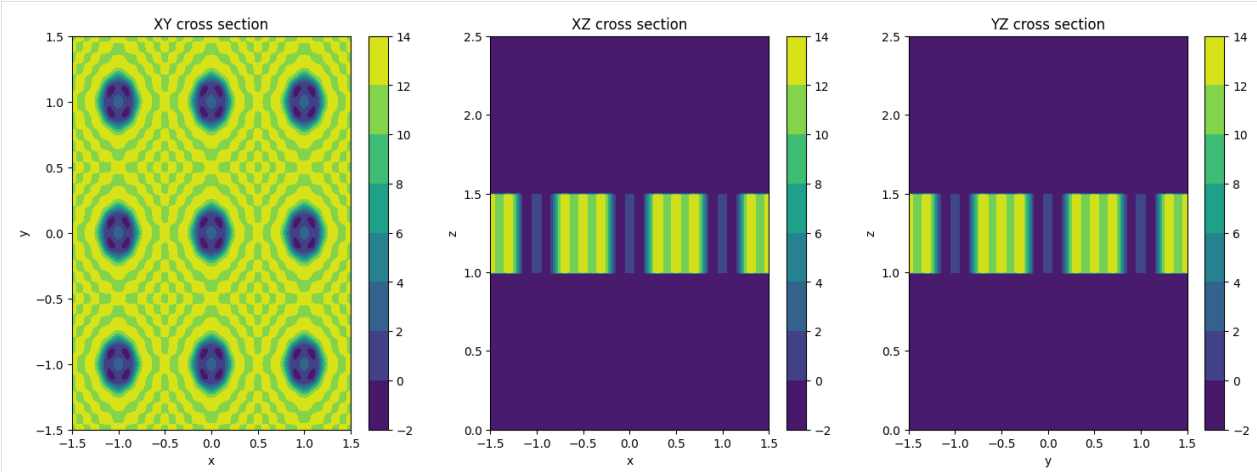
Definition of uniform layer

[4]:
```
air = A_FMM.Layer_uniform(N, N, eps=1.0)
```

Definition of 3D structure

[5]:
```
st = A_FMM.Stack(
    layers= [air, slab, air],
    d = [1.0, 0.5, 1.0],
)
_1 = np.linspace(-1.5, 1.5, 301)
_2 = np.linspace(0.0, 2.5, 251)
eps = st.calculate_epsilon(x=_1, y=_1, z=_2)
fig, ax = plt.subplots(1,3, figsize = (18,6))
_ = ax[0].contourf(eps['x'][..., 125], eps['y'][..., 125], eps['eps'][..., 125])
ax[0].set_xlabel('x'), ax[0].set_ylabel('y'), fig.colorbar(_, ax=ax[0]), ax[0].set_title(
↪'XY cross section')
ax[1].contourf(eps['x'][:, 151, :], eps['z'][:, 151, :], eps['eps'][:, 151, :])
ax[1].set_xlabel('x'), ax[1].set_ylabel('z'), fig.colorbar(_, ax=ax[1]), ax[1].set_title(
↪'XZ cross section')
ax[2].contourf(eps['y'][151, ...], eps['z'][151, ...], eps['eps'][151, ...])
ax[2].set_xlabel('y'), ax[2].set_ylabel('z'), fig.colorbar(_, ax=ax[2]), ax[2].set_title(
↪'YZ cross section')
```

[5]:
```
(Text(0.5, 0, 'y'),
 Text(0, 0.5, 'z'),
 <matplotlib.colorbar.Colorbar at 0x7fd35e83f8e0>,
 Text(0.5, 1.0, 'YZ cross section'))
```



Performing simulation

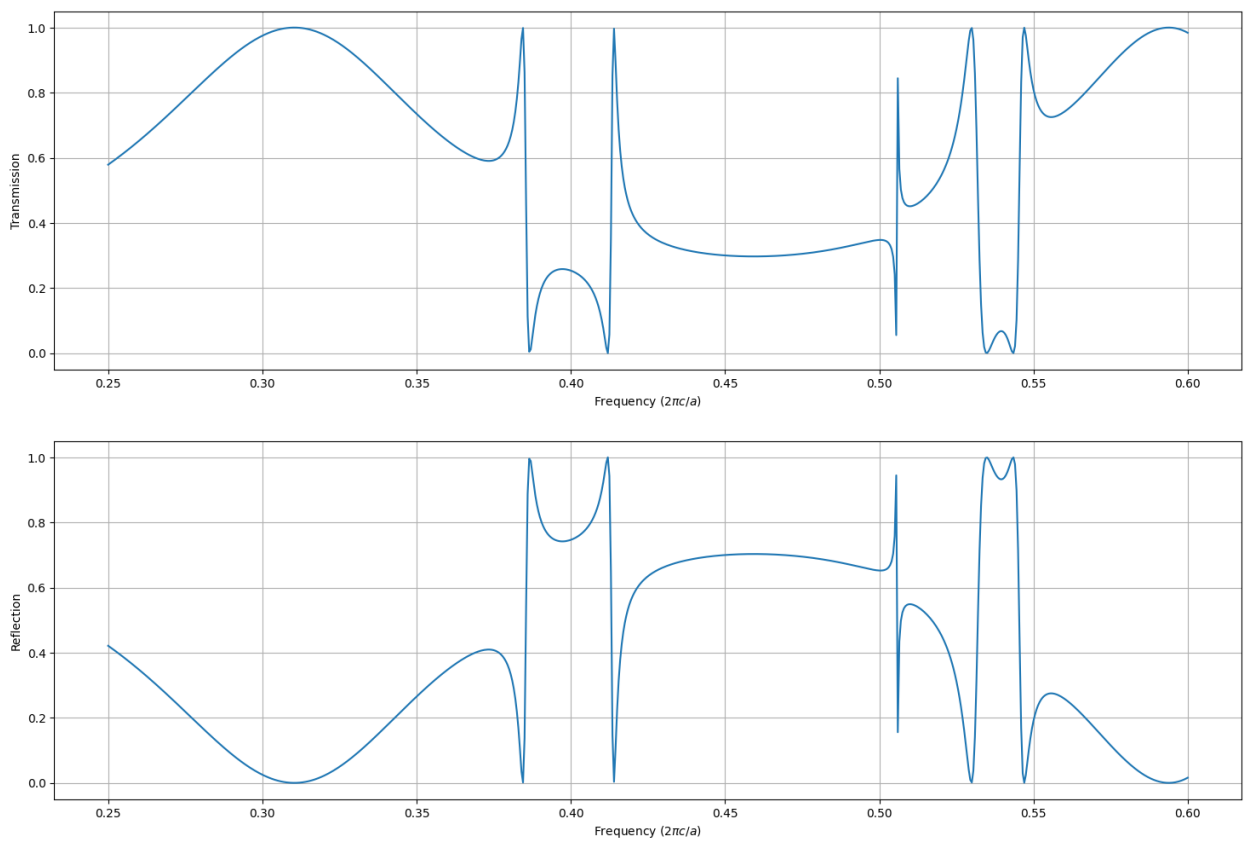
Running simulation and collecting results

[6]:
```
freqs = np.linspace(0.25, 0.6, 701)
T, R = [], []
for freq in freqs:
    st.solve(freq)
    T.append(st.get_T(0,0))
    R.append(st.get_R(0,0))
```

Plotting results

[7]:
```
fig, ax = plt.subplots(2,1, figsize = (18,12))
ax[0].plot(freqs, T)
ax[0].set_xlabel(r'Frequency ($2\pi c/a$)'), ax[0].set_ylabel('Transmission'), ax[0].
↪grid()
ax[1].plot(freqs, R)
ax[1].set_xlabel(r'Frequency ($2\pi c/a$)'), ax[1].set_ylabel('Reflection'), ax[1].grid()
```

[7]:
```
(Text(0.5, 0, 'Frequency ($2\\pi c/a$)'), Text(0, 0.5, 'Reflection'), None)
```



CHAPTER

THREE

API REFERENCE

3.1 Creator

Creator([x_list, y_list, eps_lists])	Class for the definition of the eps profile in the layer

3.1.1 A_FMM.Creator

class A_FMM.**Creator**(*x_list=None*, *y_list=None*, *eps_lists=None*)

Class for the definition of the eps profile in the layer

__init__(*x_list=None*, *y_list=None*, *eps_lists=None*)

Creator

Parameters

- **x_list** (*list*) – list of floats containig the coordinates of the x boundaries
- **y_list** (*list*) – list of floats containig the coordinates of the y boundaries
- **eps_lists** (*list*) – list of list of floats containig the eps value of the squares defined by x_list and y_list

Methods

__init__([x_list, y_list, eps_lists])	Creator
circle(e_in, e_out, r, n)	
etched_stack(eps_uc, eps_lc, w, etch, ...)	
hole(h, w, r, e_core, e_lc, e_up, e_fill)	Rib waveguide with a hole in the middle
plot_eps([N])	
rect(eps_core, eps_clad, w, h[, off_x, off_y])	Rectangular waveguide
ridge(eps_core, eps_lc, eps_uc, w, h[, t, ...])	Rib waveguide with single layer
ridge_double(eps_core, eps_lc, eps_uc, w1, ...)	Rib waveguide with double etch
ridge_pn(eps0, epsp, epsn, eps_lc, eps_uc, ...)	
slab(eps_core, eps_lc, eps_uc, w[, offset])	1D slab in x direction
slab_y(eps_core, eps_lc, eps_uc, w)	1D slab in y direction
slow_2D(eps_core, eps_c, w, Z)	
slow_general(eps_core, eps_lc, eps_uc, w, h, ...)	
varied_epi(eps_back, data_list[, y_off])	
varied_plane(eps_back, t, data_list)	
x_stack(x_l, eps_l)	

3.1.2 Methods

Creator.circle(e_in, e_out, r, n)	
Creator.etched_stack(eps_uc, eps_lc, w, ...)	
Creator.hole(h, w, r, e_core, e_lc, e_up, e_fill)	Rib waveguide with a hole in the middle
Creator.plot_eps([N])	
Creator.rect(eps_core, eps_clad, w, h[, ...])	Rectangular waveguide
Creator.ridge(eps_core, eps_lc, eps_uc, w, h)	Rib waveguide with single layer
Creator.ridge_double(eps_core, eps_lc, ...)	Rib waveguide with double etch
Creator.ridge_pn(eps0, epsp, epsn, eps_lc, ...)	
Creator.slab(eps_core, eps_lc, eps_uc, w[, ...])	1D slab in x direction
Creator.slab_y(eps_core, eps_lc, eps_uc, w)	1D slab in y direction
Creator.slow_2D(eps_core, eps_c, w, Z)	
Creator.slow_general(eps_core, eps_lc, ...)	
Creator.varied_epi(eps_back, data_list[, y_off])	
Creator.varied_plane(eps_back, t, data_list)	
Creator.x_stack(x_l, eps_l)	

A_FMM.Creator.circle

`Creator.`**`circle`**(*e_in*, *e_out*, *r*, *n*)

A_FMM.Creator.etched_stack

`Creator.`**`etched_stack`**(*eps_uc*, *eps_lc*, *w*, *etch*, *eps_stack*, *d_stack*)

A_FMM.Creator.hole

`Creator.`**`hole`**(*h*, *w*, *r*, *e_core*, *e_lc*, *e_up*, *e_fill*)

Rib waveguide with a hole in the middle

Parameters

- **h** (*TYPE*) – height of the waveguide (in unit of ay).
- **w** (*TYPE*) – width of the waveguide (in unit of ax).
- **r** (*TYPE*) – radius of the internal hole (in unit of ax).
- **e_core** (*TYPE*) – epsilon of the core.
- **e_lc** (*TYPE*) – epsilon of the lower cladding.
- **e_up** (*TYPE*) – epsilon of the upper cladding.

- **e_fill** (*TYPE*) – epsilon inside the hole.

Returns
None.

A_FMM.Creator.plot_eps

`Creator.plot_eps(N=101)`

A_FMM.Creator.rect

`Creator.rect(eps_core, eps_clad, w, h, off_x=0.0, off_y=0.0)`

Rectangular waveguide

Parameters

- **eps_core** (*float*) – epsilon of the core
- **eps_clad** (*floar*) – epsilon of the cladding
- **w** (*float*) – width of the waveguide (in unit of ax)
- **h** (*float*) – height of the waveguide (in unit of ay)
- **off_y** (*float*) – offset of the center of the waveguide with respect to the cell (in unit of ay). Default is 0.
- **off_x** (*float*) – offset of the center of the waveguide with respect to the cell (in unit of ax). Default is 0.

A_FMM.Creator.ridge

`Creator.ridge(eps_core, eps_lc, eps_uc, w, h, t=0.0, y_offset=0.0, x_offset=0.0)`

Rib waveguide with single layer

Parameters

- **eps_core** (*float*) – epsilon of the core
- **eps_lc** (*floar*) – epsilon of the lower cladding
- **eps_up** (*float*) – epsilon of the upper cladding
- **w** (*float*) – width of the rib (in unit of ax)
- **h** (*float*) – height of the un-etched part (in unit of ay)
- **t** (*float*) – height of the etched part (in unit of ay). Default is 0 (strip waveguide)
- **x_offset** (*float*) – offset of the center of the waveguide with respec to the center of the cell (in unit of ax). Default is 0
- **y_offset** (*float*) – offset of the etched part with resoect to the unetched one (in unit of ay). Default is 0 (etched and unetched part are aligned at the bottom)

A_FMM.Creator.ridge_double

`Creator.ridge_double(eps_core, eps_lc, eps_uc, w1, w2, h, t1, t2, y_offset=0.0, x_offset=0.0)`

Rib waveguide with double etch

Parameters

- **eps_core** (*float*) – epsilon of the core
- **eps_lc** (*floar*) – epsilon of the lower cladding
- **eps_up** (*float*) – epsilon of the upper cladding
- **w1** (*float*) – width of the unetched part (in unit of ax)
- **w2** (*float*) – width of the intermediate etched part (in unit of ax)
- **h** (*float*) – height of the un-etched part (in unit of ay)
- **t1** (*float*) – height of the intermidiate etched part (in unit of ay).
- **t2** (*float*) – height of the maximum etched part (in unit of ay).
- **x_offset** (*float*) – offset of the center of the waveguide with respec to the center of the cell (in unit of ax). Default is 0
- **y_offset** (*float*) – offset of the etched part with resoect to the unetched one (in unit of ay). Default is 0 (etched and unetched part are aligned at the bottom)

A_FMM.Creator.ridge_pn

`Creator.ridge_pn(eps0, epsp, epsn, eps_lc, eps_uc, w, h, t, xp, xn)`

A_FMM.Creator.slab

`Creator.slab(eps_core, eps_lc, eps_uc, w, offset=0.0)`

1D slab in x direction

Parameters

- **eps_core** (*float*) – epsilon of the core.
- **eps_lc** (*float*) – epsilon of the lower cladding.
- **eps_uc** (*float*) – epsilon of the upper cladding.
- **w** (*float*) – thickness of the slab (in unit of ax).
- **offset** (*float, optional*) – Offset if the slab with respect to the center of the cell. Defaults to 0.0.

Returns

None.

A_FMM.Creator.slab_y

`Creator.slab_y(eps_core, eps_lc, eps_uc, w)`

1D slab in y direction

Parameters

- **eps_core** (*float*) – epsilon of the core.
- **eps_lc** (*float*) – epsilon of the lower cladding.
- **eps_uc** (*float*) – epsilon of the upper cladding.
- **w** (*float*) – thickness of the slab (in unit of ay).
- **offset** (*float, optional*) – Offset if the slab with respect to the center of the cell. Defaults to 0.0.

Returns

None.

A_FMM.Creator.slow_2D

`Creator.slow_2D(eps_core, eps_c, w, Z)`

A_FMM.Creator.slow_general

`Creator.slow_general(eps_core, eps_lc, eps_uc, w, h, t, Z)`

A_FMM.Creator.varied_epi

`Creator.varied_epi(eps_back, data_list, y_off=0.0)`

A_FMM.Creator.varied_plane

`Creator.varied_plane(eps_back, t, data_list)`

A_FMM.Creator.x_stack

`Creator.x_stack(x_l, eps_l)`

3.2 Layer

Layer(Nx, Ny, creator[, Nyx])	Class for the definition of a single layer
Layer_ani_diag(Nx, Ny, creator_x, creator_y, ...)	Class for the definition of a single layer anysitropic (diagonal) layer
Layer_empty_st(Nx, Ny, creator[, Nyx])	Class for the definition of an empy layer
Layer_num(Nx, Ny, func[, args, Nyx, NX, NY])	Class for the definition of a single layer from a function defining the dielectric profile
Layer_uniform(Nx, Ny, eps[, Nyx])	Class for the definition of a single uniform layer

3.2.1 A_FMM.Layer

class A_FMM.**Layer**(*Nx: int*, *Ny: int*, *creator:* Creator, *Nyx: float = 1.0*)

Class for the definition of a single layer

__init__(*Nx: int*, *Ny: int*, *creator:* Creator, *Nyx: float = 1.0*)

Creator

Parameters

- **Nx** (*int*) – truncation order in x direction
- **Ny** (*int*) – truncation order in y direction
- **Nyx** (*float*) – ratio between the cell's dimension in y and x (ay/ax)

Methods

T_interface(lay)	Builds the Transfer matrix of the interface with another layer
T_prop(d)	Build the propagation Transfer matrix of the layer
__init__(Nx, Ny, creator[, Nyx])	Creator
add_transform_matrix([ex, FX, ey, FY])	Function for adding matrix of a coordinate transform
calculate_epsilon([x, y, z])	Return epsilon given the coordinates
calculate_field(u[, d, x, y, z, components])	Return field given modal coefficient and coordinates
calculate_field_old(u[, d, x, y, z, components])	Return field given modal coefficient and coordinates
clear()	Removes data created in mode method
coupling(u, up)	EXPERIMENTAL: Calculates coupling between two modes given their modal exapnsion
create_input(dic)	Creates the array of modal coefficient using a dictionary as input
eps_plot([pdf, N, s])	Function for plotting the dielectric consstat rebuit from plane wave expansion
get_Enorm()	Calculate field normalization
get_P_norm()	Creates array of single mode Poynting vector components.
get_Poyinting_norm()	Calculates the normalization matrix for the Poyinting vector calculations
get_Poynting(u[, d])	Calculates total Poynting vector in the layer given arrays of modal expansion
get_Poynting_single(i, u[, ordered])	Return the Poyinting vector of a single mode given the modal expansion in the layer
get_index([ordered])	Returns the effective idexes of the modes
get_input(func[, args, Nxp, Nyp, fileprint])	Expands an arbitrary fieldd shape on the basis of the layer eigenmodes
get_modal_field(i[, x, y, components])	Returns modal field profile
inspect([st])	Function for inspectig the attributes of a layer object
interface(lay)	Builds the Scattering matrix of the interface with another layer
mat_plot(name)	Plot the absolute values of the fourier trasnsform matrices
mode(k0[, kx, ky])	Calculates the eighenmode of the layer
overlap(u[, up])	EXPERIMENTAL: Calculates overlap between two fields given the modal expansion
plot_Ham(pdf)	Plot the matrix of the eigenvalue problem
transform([ex, ey, complex_transform])	Function for adding the real coordinate transfomr to the layer

3.2.2 A_FMM.Layer_ani_diag

class A_FMM.**Layer_ani_diag**(*Nx: int*, *Ny: int*, *creator_x:* Creator, *creator_y:* Creator, *creator_z:* Creator, *Nyx: float = 1.0*)

Class for the definition of a single layer anysitropic (diagonal) layer

__init__(*Nx: int*, *Ny: int*, *creator_x:* Creator, *creator_y:* Creator, *creator_z:* Creator, *Nyx: float = 1.0*)

Creator

Parameters

- **Nx** (*int*) – truncation order in x direction
- **Ny** (*int*) – truncation order in y direction
- **Nyx** (*float*) – ratio between the cell's dimension in y and x (ay/ax)

Methods

T_interface(lay)	Builds the Transfer matrix of the interface with another layer
T_prop(d)	Build the propagation Transfer matrix of the layer
__init__(Nx, Ny, creator_x, creator_y, creator_z)	Creator
add_transform_matrix([ex, FX, ey, FY])	Function for adding matrix of a coordinate transform
calculate_epsilon([x, y, z])	Return epsilon given the coordinates
calculate_field(u[, d, x, y, z, components])	Return field given modal coefficient and coordinates
calculate_field_old(u[, d, x, y, z, components])	Return field given modal coefficient and coordinates
clear()	Removes data created in mode method
coupling(u, up)	EXPERIMENTAL: Calculates coupling between two modes given their modal exapnsion
create_input(dic)	Creates the array of modal coefficient using a dictionary as input
eps_plot([pdf, N, s])	Function for plotting the dielectric consstat rebuit from plane wave expansion
get_Enorm()	Calculate field normalization
get_P_norm()	Creates array of single mode Poynting vector components.
get_Poyinting_norm()	Calculates the normalization matrix for the Poyinting vector calculations
get_Poynting(u[, d])	Calculates total Poynting vector in the layer given arrays of modal expansion
get_Poynting_single(i, u[, ordered])	Return the Poyinting vector of a single mode given the modal expansion in the layer
get_index([ordered])	Returns the effective idexes of the modes
get_input(func[, args, Nxp, Nyp, fileprint])	Expands an arbitrary fieldd shape on the basis of the layer eigenmodes
get_modal_field(i[, x, y, components])	Returns modal field profile
inspect([st])	Function for inspectig the attributes of a layer object
interface(lay)	Builds the Scattering matrix of the interface with another layer
mat_plot(name)	Plot the absolute values of the fourier trasnsform matrices
mode(k0[, kx, ky])	Calculates the eighenmode of the layer
overlap(u[, up])	EXPERIMENTAL: Calculates overlap between two fields given the modal expansion
plot_Ham(pdf)	Plot the matrix of the eigenvalue problem
transform([ex, ey, complex_transform])	Function for adding the real coordinate transfomr to the layer

3.2.3 A_FMM.Layer_empty_st

class A_FMM.**Layer_empty_st**(*Nx: int*, *Ny: int*, *creator:* Creator, *Nyx: float = 1.0*)

Class for the definition of an empy layer

__init__(*Nx: int*, *Ny: int*, *creator:* Creator, *Nyx: float = 1.0*)

Creator

Parameters

- **Nx** (*int*) – truncation order in x direction
- **Ny** (*int*) – truncation order in y direction
- **Nyx** (*float*) – ratio between the cell's dimension in y and x (ay/ax)

Methods

T_interface(lay)	Builds the Transfer matrix of the interface with another layer
T_prop(d)	Build the propagation Transfer matrix of the layer
__init__(Nx, Ny, creator[, Nyx])	Creator
add_transform_matrix([ex, FX, ey, FY])	Function for adding matrix of a coordinate transform
calculate_epsilon([x, y, z])	Return epsilon given the coordinates
calculate_field(u[, d, x, y, z, components])	Return field given modal coefficient and coordinates
calculate_field_old(u[, d, x, y, z, components])	Return field given modal coefficient and coordinates
clear()	Removes data created in mode method
coupling(u, up)	EXPERIMENTAL: Calculates coupling between two modes given their modal exapnsion
create_input(dic)	Creates the array of modal coefficient using a dictionary as input
eps_plot([pdf, N, s])	Function for plotting the dielectric consstat rebuit from plane wave expansion
fourier()	Calculates the fourier transform matrices need for the eigenvalue problem.
get_Enorm()	Calculate field normalization
get_P_norm()	Creates array of single mode Poynting vector components.
get_Poyinting_norm()	Calculates the normalization matrix for the Poyinting vector calculations
get_Poynting(u[, d])	Calculates total Poynting vector in the layer given arrays of modal expansion
get_Poynting_single(i, u[, ordered])	Return the Poyinting vector of a single mode given the modal expansion in the layer
get_index([ordered])	Returns the effective idexes of the modes
get_input(func[, args, Nxp, Nyp, fileprint])	Expands an arbitrary fieldd shape on the basis of the layer eigenmodes
get_modal_field(i[, x, y, components])	Returns modal field profile
inspect([st])	Function for inspectig the attributes of a layer object
interface(lay)	Builds the Scattering matrix of the interface with another layer
mat_plot(name)	Plot the absolute values of the fourier trasnsform matrices
mode(k0[, kx, ky])	Calculates the eighenmode of the layer
overlap(u[, up])	EXPERIMENTAL: Calculates overlap between two fields given the modal expansion
plot_Ham(pdf)	Plot the matrix of the eigenvalue problem
transform([ex, ey, complex_transform])	Function for adding the real coordinate transfomr to the layer

3.2.4 A_FMM.Layer_num

class A_FMM.**Layer_num**(*Nx: int*, *Ny: int*, *func: callable*, *args: tuple = None*, *Nyx: float = 1.0*, *NX: int = 2048*, *NY: int = 2048*)

Class for the definition of a single layer from a function defining the dielectric profile

__init__(*Nx: int*, *Ny: int*, *func: callable*, *args: tuple = None*, *Nyx: float = 1.0*, *NX: int = 2048*, *NY: int = 2048*)

Creator

Parameters

- **Nx** (*int*) – truncation order in x direction
- **Ny** (*int*) – truncation order in y direction
- **Nyx** (*float*) – ratio between the cell's dimension in y and x (ay/ax)

Methods

T_interface(lay)	Builds the Transfer matrix of the interface with another layer
T_prop(d)	Build the propagation Transfer matrix of the layer
__init__(Nx, Ny, func[, args, Nyx, NX, NY])	Creator
add_transform_matrix([ex, FX, ey, FY])	Function for adding matrix of a coordinate transform
calculate_epsilon([x, y, z])	Return epsilon given the coordinates
calculate_field(u[, d, x, y, z, components])	Return field given modal coefficient and coordinates
calculate_field_old(u[, d, x, y, z, components])	Return field given modal coefficient and coordinates
clear()	Removes data created in mode method
coupling(u, up)	EXPERIMENTAL: Calculates coupling between two modes given their modal exapnsion
create_input(dic)	Creates the array of modal coefficient using a dictionary as input
eps_plot([pdf, N, s])	Function for plotting the dielectric consstat rebuit from plane wave expansion
get_Enorm()	Calculate field normalization
get_P_norm()	Creates array of single mode Poynting vector components.
get_Poyinting_norm()	Calculates the normalization matrix for the Poyinting vector calculations
get_Poynting(u[, d])	Calculates total Poynting vector in the layer given arrays of modal expansion
get_Poynting_single(i, u[, ordered])	Return the Poyinting vector of a single mode given the modal expansion in the layer
get_index([ordered])	Returns the effective idexes of the modes
get_input(func[, args, Nxp, Nyp, fileprint])	Expands an arbitrary fieldd shape on the basis of the layer eigenmodes
get_modal_field(i[, x, y, components])	Returns modal field profile
inspect([st])	Function for inspectig the attributes of a layer object
interface(lay)	Builds the Scattering matrix of the interface with another layer
mat_plot(name)	Plot the absolute values of the fourier trasnsform matrices
mode(k0[, kx, ky])	Calculates the eighenmode of the layer
overlap(u[, up])	EXPERIMENTAL: Calculates overlap between two fields given the modal expansion
plot_Ham(pdf)	Plot the matrix of the eigenvalue problem
transform([ex, ey, complex_transform])	Function for adding the real coordinate transfomr to the layer

3.2.5 A_FMM.Layer_uniform

class A_FMM.**Layer_uniform**(*Nx: int*, *Ny: int*, *eps: float | complex*, *Nyx: float = 1.0*)

Class for the definition of a single uniform layer

__init__(*Nx: int*, *Ny: int*, *eps: float | complex*, *Nyx: float = 1.0*)

Creator

Parameters

- **Nx** (*int*) – truncation order in x direction
- **Ny** (*int*) – truncation order in y direction
- **Nyx** (*float*) – ratio between the cell's dimension in y and x (ay/ax)

Methods

T_interface(lay)	Builds the Transfer matrix of the interface with another layer
T_prop(d)	Build the propagation Transfer matrix of the layer
__init__(Nx, Ny, eps[, Nyx])	Creator
add_transform_matrix([ex, FX, ey, FY])	Function for adding matrix of a coordinate transform
calculate_epsilon([x, y, z])	Return epsilon given the coordinates
calculate_field(u[, d, x, y, z, components])	Return field given modal coefficient and coordinates
calculate_field_old(u[, d, x, y, z, components])	Return field given modal coefficient and coordinates
clear()	Removes data created in mode method
coupling(u, up)	EXPERIMENTAL: Calculates coupling between two modes given their modal exapnsion
create_input(dic)	Creates the array of modal coefficient using a dictionary as input
eps_plot([pdf, N, s])	Function for plotting the dielectric consstat rebuit from plane wave expansion
get_Enorm()	Calculate field normalization
get_P_norm()	Creates array of single mode Poynting vector components.
get_Poyinting_norm()	Calculates the normalization matrix for the Poyinting vector calculations
get_Poynting(u[, d])	Calculates total Poynting vector in the layer given arrays of modal expansion
get_Poynting_single(i, u[, ordered])	Return the Poyinting vector of a single mode given the modal expansion in the layer
get_index([ordered])	Returns the effective idexes of the modes
get_input(func[, args, Nxp, Nyp, fileprint])	Expands an arbitrary fieldd shape on the basis of the layer eigenmodes
get_modal_field(i[, x, y, components])	Returns modal field profile
inspect([st])	Function for inspectig the attributes of a layer object
interface(lay)	Builds the Scattering matrix of the interface with another layer
mat_plot(name)	Plot the absolute values of the fourier trasnsform matrices
mode(k0[, kx, ky])	Calculates the eighenmode of the layer
overlap(u[, up])	EXPERIMENTAL: Calculates overlap between two fields given the modal expansion
plot_Ham(pdf)	Plot the matrix of the eigenvalue problem
transform([ex, ey, complex_transform])	Function for adding the real coordinate transfomr to the layer

3.2.6 Methods

Layer._check_array_shapes(u, d)	Chekcs that the modal amplitudea arrays and the coordinates arrays have consistent shapes
Layer._filter_componets([components])	Checks if the fileds components list contains only allowed ones
Layer._process_xy(x, y)	Transform the x and y coordinates between the real and computational space
Layer.add_transform_matrix([ex, FX, ey, FY])	Function for adding matrix of a coordinate transform
Layer.calculate_epsilon([x, y, z])	Return epsilon given the coordinates
Layer.calculate_field(u[, d, x, y, z, ...])	Return field given modal coefficient and coordinates
Layer.calculate_field_old(u[, d, x, y, z, ...])	Return field given modal coefficient and coordinates
Layer.clear()	Removes data created in mode method
Layer.coupling(u, up)	EXPERIMENTAL: Calculates coupling between two modes given their modal exapnsion
Layer.create_input(dic)	Creates the array of modal coefficient using a dictionary as input
Layer.eps_plot([pdf, N, s])	Function for plotting the dielectric consstat rebuit from plane wave expansion
Layer.get_Enorm()	Calculate field normalization
Layer.get_P_norm()	Creates array of single mode Poynting vector components.
Layer.get_Poyinting_norm()	Calculates the normalization matrix for the Poyinting vector calculations
Layer.get_Poynting(u[, d])	Calculates total Poynting vector in the layer given arrays of modal expansion
Layer.get_Poynting_single(i, u[, ordered])	Return the Poyinting vector of a single mode given the modal expansion in the layer
Layer.get_index([ordered])	Returns the effective idexes of the modes
Layer.get_input(func[, args, Nxp, Nyp, ...])	Expands an arbitrary fieldd shape on the basis of the layer eigenmodes
Layer.get_modal_field(i[, x, y, components])	Returns modal field profile
Layer.inspect([st])	Function for inspectig the attributes of a layer object
Layer.interface(lay)	Builds the Scattering matrix of the interface with another layer
Layer.mat_plot(name)	Plot the absolute values of the fourier trasnsform matrices
Layer.mode(k0[, kx, ky])	Calculates the eighenmode of the layer
Layer.overlap(u[, up])	EXPERIMENTAL: Calculates overlap between two fields given the modal expansion
Layer.plot_Ham(pdf)	Plot the matrix of the eigenvalue problem
Layer.transform([ex, ey, complex_transform])	Function for adding the real coordinate transfomr to the layer

A_FMM.Layer._check_array_shapes

static Layer.**_check_array_shapes**(*u: ndarray*, *d: ndarray*) → None

Chekcs that the modal amplitudea arrays and the coordinates arrays have consistent shapes

A_FMM.Layer._filter_componets

static Layer.**_filter_componets**(*components: list = None*) → list

Checks if the fileds components list contains only allowed ones

A_FMM.Layer._process_xy

Layer.**_process_xy**(*x: ndarray*, *y: ndarray*) → tuple

Transform the x and y coordinates between the real and computational space

Parameters

- **x** (*ndarray*) – array of x coordinates in the real space
- **y** (*ndarray*) – array of y coordinates in the real space

Returns: tuple of numpy.ndarray containing:

- ndarray: array of x coordinates in the computational space
- ndarray: array of y coordinates in the computational space

A_FMM.Layer.add_transform_matrix

Layer.**add_transform_matrix**(*ex: float = 0.0*, *FX: ndarray = None*, *ey: float = 0.0*, *FY: ndarray = None*)

Function for adding matrix of a coordinate transform

Parameters

- **ex** (*float*) – relative width of the unmapped region in x direction. Default is 0. This is only for keeping track of the value, as it has no effect on the transformation.
- **FX** (*ndarray*) – FX matrix of the coordinate trasnformation
- **ey** (*float*) – relative width of the unmapped region in y direction. Default is 0. This is only for keeping track of the value, as it has no effect on the transformation.
- **FY** (*ndarray*) – FY matrix of the coordinate trasnformation

A_FMM.Layer.calculate_epsilon

Layer.**calculate_epsilon**(*x: ndarray = 0.0*, *y: ndarray = 0.0*, *z: ndarray = 0.0*) → dict[str, ndarray]

Return epsilon given the coordinates

The epsilon returned here is the one reconstructed from the Fourier transform. The epsilon is reconstructed on the meshgrid of x,y, and z.

Parameters

- **x** (*array_like*) – x coordinates (1D array).
- **y** (*array_like*) – y coordinates (1D array).

- **z** (*array_like*) – z coordinates (1D array).

Returns
Epsilon value at coordinates. Shape of ndarray is the same as x,y, and z.

Return type
ndarray

Raises
ValueError – if x,y,z have different shapes

A_FMM.Layer.calculate_field

`Layer.calculate_field(u: ndarray, d: ndarray = None, x: ndarray = 0, y: ndarray = 0, z: ndarray = 0, components: list = None) → dict`

Return field given modal coefficient and coordinates

Coordinates arrays must be 1D. Fields are returned on a meshgrid of the input coordinates.

Parameters

- **u** (*array_like*) – coefficient of forward propagating modes.
- **d** (*array_like, optional*) – coefficient of backward propagating modes. Default to None: no backward propagation is assumed.
- **x** (*array_like*) – x coordinates.
- **y** (*array_like*) – y coordinates.
- **z** (*array_like*) – z coordinates.
- **components** (*list of str, optional*) – field components to calculate. Default to None: all components ('Ex', 'Ey', 'Hx', 'Hy') are calculated.

Returns
Desired field components. Shape of ndarray is the same as x,y, and z.

Return type
dict of ndarray

Raises
ValueError – if other component than 'Ex', 'Ey', 'Hx', or 'Hy' is requested.

A_FMM.Layer.calculate_field_old

`Layer.calculate_field_old(u: ndarray, d: ndarray = None, x: ndarray = 0, y: ndarray = 0, z: ndarray = 0, components: list = None) → dict`

Return field given modal coefficient and coordinates

Coordinates arrays must be 1D. Fields are returned on a meshgrid of the input coordinates. Older version. Slower, but may require less memory.

Parameters

- **u** (*array_like*) – coefficient of forward propagating modes.
- **d** (*array_like, optional*) – coefficient of backward propagating modes. Default to None: no backward propagation is assumed.
- **x** (*array_like*) – x coordinates.

- **y** (*array_like*) – y coordinates.
- **z** (*array_like*) – z coordinates.
- **components** (*list of str, optional*) – field components to calculate. Default to None: all components ('Ex', 'Ey', 'Hx', 'Hy') are calculated.

Returns
Desired field components. Shape of ndarray is the same as x,y, and z.

Return type
dict of ndarray

Raises
ValueError – if other component than 'Ex', 'Ey', 'Hx', or 'Hy' is requested.

A_FMM.Layer.clear

`Layer.clear()`

Removes data created in mode method

A_FMM.Layer.coupling

`Layer.coupling(u: ndarray, up: ndarray)` → tuple

EXPERIMENTAL: Calculates coupling between two modes given their modal exapnsion

Parameters

- **u** (*TYPE*) – Modal coefficient of first mode.
- **up** (*TYPE*) – Modal coefficient of second mode.

Returns
[tx, tx]: floats. Coupling in x and y polarization.

Return type
list

A_FMM.Layer.create_input

`Layer.create_input(dic: dict)` → ndarray

Creates the array of modal coefficient using a dictionary as input

Parameters
dic (*dict*) – Dictionary of exited modes {modal_index : modal_coeff}. Modes are ordered.

Returns
Array of modal coefficient.

Return type
u (1darray)

A_FMM.Layer.eps_plot

`Layer.eps_plot(pdf=None, N=200, s=1)`

Function for plotting the dielectric consstat rebuit from plane wave expansion

Parameters

- **pdf** (*string or PdfPages*) – file for printing the the epsilon if a PdfPages object, the page is appended to the pdf if string, a pdf with that name is created
- **N** (*int*) – number of points
- **s** (*float*) – number of cell replicas to display (default 1)

A_FMM.Layer.get_Enorm

`Layer.get_Enorm()`

Calculate field normalization

Returns
None.

A_FMM.Layer.get_P_norm

`Layer.get_P_norm()`

Creates array of single mode Poynting vector components.

It is stored in the P_norm attribute

Returns
None.

A_FMM.Layer.get_Poyinting_norm

`Layer.get_Poyinting_norm()`

Calculates the normalization matrix for the Poyinting vector calculations

Returns
None.

A_FMM.Layer.get_Poynting

`Layer.get_Poynting(u: ndarray, d: ndarray = None)`

Calculates total Poynting vector in the layer given arrays of modal expansion

Parameters

- **u** (*1darray*) – Modal expansion of forward propagating modes.
- **d** (*1darray, optional*) – Modal expansion of backward propagating modes. Defaults to None.

Returns
DESCRIPTION.

Return type
TYPE

A_FMM.Layer.get_Poynting_single

`Layer.get_Poynting_single(i: int, u: ndarray, ordered: bool = True) → float`

Return the Poyinting vector of a single mode given the modal expansion in the layer

Parameters

- **i** (*int*) – Index of the mode.
- **u** (*1darray*) – Array of modal coefficient.
- **ordered** (*TYPE, optional*) – Regulates how mode are ordered. If True, they are ordered for decreasing effective index. If Flase, the order is whatever is returned by the diagonalization routine. Defaults to True.

Returns
DESCRIPTION.

Return type
TYPE

A_FMM.Layer.get_index

`Layer.get_index(ordered: bool = True) → ndarray`

Returns the effective idexes of the modes

Parameters
ordered (*bool*) – if True (default) the modes are ordered by decreasing effective index

A_FMM.Layer.get_input

`Layer.get_input(func: callable, args: tuple = None, Nxp: int = 1024, Nyp: int = None, fileprint: str = None) → ndarray`

Expands an arbitrary fieldd shape on the basis of the layer eigenmodes

Parameters

- **func** (*function*) – Function describing the field. This function should be in the form (x,y,*args). It must be able to accept x and y as numpy array. It must return two values, expressing Ex and Ey
- **args** (*tuple, optional*) – Eventual tuple of additional arguments for func. Defaults to None.
- **Nxp** (*int, optional*) – Number of points to evaluate the function in the x direction. Defaults to 1024.
- **Nyp** (*int, optional*) – Number of points to evaluate the function in the y direction. Defaults to None (1 if layer is 1D, Nxp if 2D).
- **fileprint** (*str, optional*) – Filename on which to write the used function. Mainly for debug. Defaults to None.

Returns
Array of the modal coefficient of the expansion.

Return type
u (1darray)

A_FMM.Layer.get_modal_field

`Layer.get_modal_field(i: int, x: float = 0.0, y: float = 0.0, components: list = None)` → dict

Returns modal field profile

Parameters

- **i** (*int*) – index of the mode.
- **x** (*float or array_like*) – x coordinate for the field calculation
- **y** (*float or array_like*) – y coordinate for the field calculation
- **components** (*list of str, optional*) – field components to calculate. Default to None: all components ('Ex', 'Ey', 'Hx', 'Hy') are calculated.

Returns
modal field

Return type
DataFrame

A_FMM.Layer.inspect

`Layer.inspect(st='')`

Function for inspectig the attributes of a layer object

Parameters
st (*string*) – string to print before the inspection for identification

A_FMM.Layer.interface

`Layer.interface(lay)` → *S_matrix*

Builds the Scattering matrix of the interface with another layer

Parameters
lay (Layer) – Layer toward which to calculate the scattering matrix.

Returns
Interface scattering matrix.

Return type
S (*S_matrix*)

A_FMM.Layer.mat_plot

`Layer.mat_plot(name: str)`

Plot the absolute values of the fourier trasnsform matrices

Parameters

- **name** (*str*) – name of the pdf file for plotting
- **N** (*int*) – number of points for plotting the epsilon
- **s** (*float*) – number pf relicas of the cell to plot. Default is 1.

A_FMM.Layer.mode

`Layer.mode(k0: float, kx: float = 0.0, ky: float = 0.0)`

Calculates the eighenmode of the layer

Parameters

- **k0** (*float*) – Vacuum wavevector
- **kx** (*float*) – Wavevector in the x direction
- **ky** (*float*) – Wavevector in the y direction

A_FMM.Layer.overlap

`Layer.overlap(u: ndarray, up: ndarray = None)`

EXPERIMENTAL: Calculates overlap between two fields given the modal expansion

Parameters

- **u** (*1darray*) – Modal coefficient of first mode.
- **up** (*1darray, optional*) – Modal coefficient of first mode. Defaults to None (up=u, namely normalization is returned).

Returns

[tx, tx]: floats. Namely overlap in x and y polarization

Return type

list

A_FMM.Layer.plot_Ham

`Layer.plot_Ham(pdf: PdfPages) → None`

Plot the matrix of the eigenvalue problem

Parameters

pdf (*PdfPages*) – pdf object to be used to plot.

Returns

None.

A_FMM.Layer.transform

`Layer.transform(ex: float = 0, ey: float = 0, complex_transform: bool = False)`

Function for adding the real coordinate transfomr to the layer

Note: for no mapping, set the width to 0

Parameters

- **ex** (*float*) – relative width of the unmapped region in x direction. Default is 0 (no mapping)
- **ey** (*float*) – relative width of the unmapped region in y direction. Default is 0 (no mapping)
- **complex_transform** (*bool*) – False for real transform (default), True for complex one.

3.3 Stack

Stack([layers, d])	Class representing the multylayer object

3.3.1 A_FMM.Stack

class A_FMM.**Stack**(*layers: list[Layer] = None*, *d: list[float] = None*)

Class representing the multylayer object

This class is used for the definition of the multilayer object to be simulated using fourier expansion (x and y axis) and scattering matrix algorithm (z axis). It is built from a list of layers and thicknesses. The value of the thickness of the first and last layer is irrelevant for the simulation, and it is used only to set the plotting window.

__init__(*layers: list[Layer] = None*, *d: list[float] = None*) → None

Creator

Parameters

- **layers** (*list, optional*) – List of Layers: layers of the multylayer. Defaults to None (empty list).
- **d** (*list, optional*) – List of float: thicknesses of the multylayer. Defaults to None (empty list).

Raises

ValueError – Raised if d and mat have different lengths

Returns

None.

Methods

`__init__`([layers, d])	Creator
`add_layer`(lay, d)	Add a layer at the end of the multilayer
`bloch_modes`()	Calculates Bloch modes of the stack.
`calculate_epsilon`([x, y, z])	Returns epsilon in the stack
`calculate_fields`(u1[, d2, x, y, z, components])	Returns fields in the stack
`count_interface`()	Helper function to identify the different layers and the needed interfaces
`double`()	Compose the scattering matrix of the stack with itself, doubling the structure
`flip`()	Flip a solved stack
`get_PR`(i, j[, ordered])	Get phase of the relfection coefficient between modes
`get_PT`(i, j[, ordered])	Get phase of the transmission coefficient between modes
`get_R`(i, j[, ordered])	Get relfection coefficient between modes
`get_T`(i, j[, ordered])	Get transmission coefficient between modes.
`get_el`(sel, i, j)	Returns element of the scattering matrix
`get_energybalance`(u[, d])	Get total energy balance of the stack given the inputs
`get_inout`(u[, d])	Return data about the output of the structure given the input
`get_prop`(u, list_lay[, d])	Calculates the total poyinting vector in the requiested layers
`inspect`([st, details])	Print some info about the Stack
`join`(st2)	Join the scattering matrix of the structure with the one of a second structure
`loop_intermediate`(u1, d2)	Generator for the intermedia modal coefficients.
`solve`(k0[, kx, ky])	Calculates the scattering matrix of the multilayer (cpu friendly version)
`solve_S`()	Builds the scattering matrix of the stacks.
`solve_lay`(k0[, kx, ky])	Solve the eigenvalue problem of all the layer in the stack
`solve_serial`(k0[, kx, ky])	Calculates the scattering matrix of the multilayer (memory friendly version)
`transform`([ex, ey, complex_transform])	Function for adding the real coordinate transform to all layers of the stack

Attributes

`total_length`

3.3.2 Methods

Stack.add_layer(lay, d)	Add a layer at the end of the multilayer
Stack.bloch_modes()	Calculates Bloch modes of the stack.
Stack.calculate_epsilon([x, y, z])	Returns epsilon in the stack
Stack.calculate_fields(u1[, d2, x, y, z, ...])	Returns fields in the stack
Stack.count_interface()	Helper function to identify the different layers and the needed interfaces
Stack.double()	Compose the scattering matrix of the stack with itself, doubling the structure
Stack.flip()	Flip a solved stack
Stack.get_PR(i, j[, ordered])	Get phase of the relfection coefficient between modes
Stack.get_PT(i, j[, ordered])	Get phase of the transmission coefficient between modes
Stack.get_R(i, j[, ordered])	Get relfection coefficient between modes
Stack.get_T(i, j[, ordered])	Get transmission coefficient between modes.
Stack.get_el(sel, i, j)	Returns element of the scattering matrix
Stack.get_energybalance(u[, d])	Get total energy balance of the stack given the inputs
Stack.get_inout(u[, d])	Return data about the output of the structure given the input
Stack.get_prop(u, list_lay[, d])	Calculates the total poyinting vector in the requiested layers
Stack.inspect([st, details])	Print some info about the Stack
Stack.join(st2)	Join the scattering matrix of the structure with the one of a second structure
Stack.loop_intermediate(u1, d2)	Generator for the intermedia modal coefficients.
Stack.solve(k0[, kx, ky])	Calculates the scattering matrix of the multilayer (cpu friendly version)
Stack.solve_S()	Builds the scattering matrix of the stacks.
Stack.solve_lay(k0[, kx, ky])	Solve the eigenvalue problem of all the layer in the stack
Stack.solve_serial(k0[, kx, ky])	Calculates the scattering matrix of the multilayer (memory friendly version)
Stack.transform([ex, ey, complex_transform])	Function for adding the real coordinate transform to all layers of the stack

A_FMM.Stack.add_layer

`Stack.`**`add_layer`**(*lay:* Layer, *d: float*) → None

Add a layer at the end of the multilayer

Parameters

- **lay** (Layer) – Layer to be added.
- **d** (*float*) – thickness of the layer.

Returns

None.

A_FMM.Stack.bloch_modes

`Stack.bloch_modes()` → ndarray

Calculates Bloch modes of the stack.

This function assumens the stack to represent the unit cell of a periodic structure, and calculates the corresponding Bloch modes. The thickness of the first and last layer are ignored (assumed 0). To work correctly first and last layer needs to be the same.

Returns
DESCRIPTION.

Return type
TYPE

A_FMM.Stack.calculate_epsilon

`Stack.calculate_epsilon(x: ndarray = 0.0, y: ndarray = 0.0, z: ndarray = 0.0)` → dict[str, ndarray]

Returns epsilon in the stack

Epsilon is calculated on a meshgrdi of x,y,z

Parameters

- **x** (*np.ndarray*) – x coordinate (1D array)
- **y** (*np.ndarray*) – y coordinate (1D array)
- **z** (*np.ndarray*) – z coordinate (1D array)

Returs:
dict: Dictionary containing the coordinates and the epsilon

A_FMM.Stack.calculate_fields

`Stack.calculate_fields(u1: ndarray, d2: ndarray = None, x: ndarray = 0, y: ndarray = 0, z: ndarray = 0, components: list[str] = None)` → dict[str, ndarray]

Returns fields in the stack

The fields are calculated on a meshgrdi of x,y,z

Parameters

- **u1** (*np.ndarray*) – forward modal coefficient in the first layer
- **d2** (*np.ndarray*) – backward modal coefficient in the last layer
- **x** (*np.ndarray*) – x coordinate (1D array)
- **y** (*np.ndarray*) – y coordinate (1D array)
- **z** (*np.ndarray*) – z coordinate (1D array)
- **components** (*list*) – List of modal componets to be calculated. Possible are ['Ex', 'Ey', 'Hx', 'Hz']. Default to None (all of them).

Returs:
dict: Dictionary containing the coordinates and the field components

A_FMM.Stack.count_interface

`Stack.count_interface() → None`

Helper function to identify the different layers and the needed interfaces

Returns
None.

A_FMM.Stack.double

`Stack.double() → None`

Compose the scattering matrix of the stack with itself, doubling the structure

When doing this, the lenght of the first al last layer are ignored (set to 0). To function properly hoever they need to be equal (but do not need to have physical meaning)

Raises
RuntimeError – Raised if the stack is not solved yet.

Returns
None.

A_FMM.Stack.flip

`Stack.flip() → None`

Flip a solved stack

Flip the stack, swapping the left and right side

Raises
RuntimeError – Raised if the structure is not solved yet.

Returns
None.

A_FMM.Stack.get_PR

`Stack.get_PR(i: int, j: int, ordered: bool = True) → float`

Get phase of the relfection coefficient between modes

Parameters

- **i** (*int*) – Index of the source mode.
- **j** (*int*) – Index of the target mode.
- **ordered** (*bool, optional*) – If True, modes are ordered for decrasing effective index, otherwise the order is whatever is returned by the diagonalization routine. Defaults to True.

Returns
Phase of reflection between the modes

Return type
float

A_FMM.Stack.get_PT

`Stack.get_PT(i: int, j: int, ordered: bool = True) → float`

Get phase of the transmission coefficient between modes

Parameters

- **i** (*int*) – Index of the source mode.
- **j** (*int*) – Index of the target mode.
- **ordered** (*bool, optional*) – If True, modes are ordered for decrasing effective index, otherwise the order is whatever is returned by the diagonalization routine. Defaults to True.

Returns
Phase of transmission between the modes

Return type
float

A_FMM.Stack.get_R

`Stack.get_R(i: int, j: int, ordered: bool = True) → float`

Get relfection coefficient between modes

Parameters

- **i** (*int*) – Index of the source mode.
- **j** (*int*) – Index of the target mode.
- **ordered** (*bool, optional*) – If True, modes are ordered for decrasing effective index, otherwise the order is whatever is returned by the diagonalization routine. Defaults to True.

Returns
Reflection between the modes

Return type
float

A_FMM.Stack.get_T

`Stack.get_T(i: int, j: int, ordered: bool = True) → float`

Get transmission coefficient between modes.

Parameters

- **i** (*int*) – Index of the source mode.
- **j** (*int*) – Index of the target mode.
- **ordered** (*bool, optional*) – If True, modes are ordered for decrasing effective index, otherwise the order is whatever is returned by the diagonalization routine. Defaults to True.

Returns
Transmission between the modes.

Return type
float

A_FMM.Stack.get_el

`Stack.get_el(sel: str, i: int, j: int)` → complex

Returns element of the scattering matrix

Note: Modes are ordered for decrasing effective index

Parameters

- **sel** (*str*) – First index of the matrix.
- **i** (*int*) – Second index of the matrix.
- **j** (*int*) – Select the relevand submatrix. Choises are '11', '12', '21', '22'.

Raises
ValueError – If sel in not in the allowed.

Returns
Element of the scattering matrix.

Return type
complex

A_FMM.Stack.get_energybalance

`Stack.get_energybalance(u: ndarray, d: ndarray = None)` → tuple[float]

Get total energy balance of the stack given the inputs

Return total power reflected, transmitted and absorbed, normalized to the incidenc power.

Parameters

- **u** (*1darray*) – Modal coefficient of the left input.
- **d** (*1darray, optional*) – Modal coefficent of the right input. Defaults to None.

Returns

tuple contining tree floats with meaning:

- Total power out from left side (reflection if only u).
- Total power out from right side (transmission if only u).
- Total power absorbed in the stack.

Return type
tuple

A_FMM.Stack.get_inout

`Stack.get_inout(u: ndarray, d: ndarray = None)` → dict[str, tuple[ndarray, ndarray, float]]

Return data about the output of the structure given the input

Parameters

- **u** (*1darray*) – Vector of the modal coefficents of the right inputs.
- **d** (*1darray, optional*) – Vector of the modal coefficents of the right inputs. Defaults to None.

Returns

Dictionary containing data of the output:

- 'left' : (u,d,P): forward modal coefficient, backward modal coefficient and Poyinting vector at the left side.
- 'right' : (u,d,P): forward modal coefficient, backward modal coefficient and Poyinting vector at the right side.

Return type
dict

A_FMM.Stack.get_prop

`Stack.get_prop`(*u: ndarray*, *list_lay: list[int]*, *d: ndarray = None*) → dict[int, float]

Calculates the total poyinting vector in the requiested layers

Parameters

- **u** (*ndarray*) – array containing the modal coefficient incoming in the first layer.
- **list_lay** (*list of int*) – indexes of the layer of which to calculate the Poynting vector.
- **d** (*ndarray, optional*) – array containing the modal coefficient incoming in the last layer. Defaults to None.

Returns
Dictionary of the Poyting vectors in the the layers {layer_index : Poyting vector}

Return type
dic (dict)

A_FMM.Stack.inspect

`Stack.inspect`(*st: str = ''*, *details: str = 'no'*) → None

Print some info about the Stack

A_FMM.Stack.join

`Stack.join`(*st2: Self*) → None

Join the scattering matrix of the structure with the one of a second structure

When doing this, the lenght of the first al last layeror each stack are ignored (set to 0). To function last layer of self and first of st2 need to be equal (but do not need to have physical meaning). The condiction used to previoselt solve the stack needs to be the same. This is not checked by the code, so be careful.

Parameters
st2 (Stack) – Stack to which to join self.

Raises
RuntimeError – Raised is one the structure is not solved yet.

Returns
None.

A_FMM.Stack.loop_intermediate

`Stack.loop_intermediate(u1: ndarray, d2: ndarray) → tuple`

Generator for the intermedia modal coefficients.

Progressively yields the forward and backward modal coefficient given the external excitation.

Parameters

- **u1** (*np.ndarray*) – forward modal coefficient of the first layer (near the interface)
- **d2** ((*np.ndarray*) – backward modal coefficient of the last layer (near the interface)

Yields

np.ndarray – forward modal amplitudes of layer np.ndarray: backward modal amplitudes for layer Layer: layer object float: thickness of the layer

A_FMM.Stack.solve

`Stack.solve(k0: float, kx: float = 0.0, ky: float = 0.0) → None`

Calculates the scattering matrix of the multilayer (cpu friendly version)

This version of solve solve the system in the "smart" way, solving fisrt the eigenvalue problem in each unique layer and the interface matrices of all the interface involved. The computaitonal time scales with the number of different layers, not with the total one. It prioritize minimize the calculation done while using more memory.

Parameters

- **k0** (*float*) – Vacuum wavevector for the simulation (freqency).
- **kx** (*float, optional*) – Wavevector in the x direction for the pseudo-fourier expansion. Defaults to 0.0.
- **ky** (*float, optional*) – Wavevector in the x direction for the pseudo-fourier expansion. Defaults to 0.0.

Returns

None.

A_FMM.Stack.solve_S

`Stack.solve_S() → None`

Builds the scattering matrix of the stacks. It assumes that all the layers are alredy solved.

Returns

None.

A_FMM.Stack.solve_lay

`Stack.solve_lay(k0: float, kx: float = 0.0, ky: float = 0.0) → None`

Solve the eigenvalue problem of all the layer in the stack

Parameters

- **k0** (*float*) – Vacuum wavevector for the simulation (freqency).
- **kx** (*float, optional*) – Wavevector in the x direction for the pseudo-fourier expansion. Defaults to 0.0.

- **ky** (*float, optional*) – Wavevector in the x direction for the pseudo-fourier expansion. Defaults to 0.0.

Returns
None.

A_FMM.Stack.solve_serial

`Stack.solve_serial(k0: float, kx: float = 0.0, ky: float = 0.0) → None`

Calculates the scattering matrix of the multilayer (memory friendly version)

This version solves sequentially the layers and the interface as they are in the stack. It is more momery efficient since onlt the data of 2 layer are kept in memory at any given time. Computational time scales with the total number of layer, regardless if they are equal or not. It prioritize memory efficiency while possibly requiring more calculations.

Parameters

- **k0** (*float*) – Vacuum wavevector for the simulation (freqency).
- **kx** (*float, optional*) – Wavevector in the x direction for the pseudo-fourier expansion. Defaults to 0.0.
- **ky** (*float, optional*) – Wavevector in the x direction for the pseudo-fourier expansion. Defaults to 0.0.

Returns
None.

A_FMM.Stack.transform

`Stack.transform(ex: float = 0, ey: float = 0, complex_transform: bool = False) → tuple[ndarray]`

Function for adding the real coordinate transform to all layers of the stack

Note: for no mapping, set the width to 0

Parameters

- **ex** (*float*) – relative width of the unmapped region in x direction. Default is 0 (no mapping)
- **ey** (*float*) – relative width of the unmapped region in y direction. Default is 0 (no mapping)
- **complex_transform** (*bool*) – False for real transform (default), True for complex one.

3.4 S_matrix

S_matrix(N)	Implementation of the scattring matrix object

3.4.1 A_FMM.S_matrix

class A_FMM.**S_matrix**(*N*)

Implementation of the scattring matrix object

This object is a container for NxN matrices, conventionally defined as S11, S12, S21 and S22 Also, it implementens all the methods involving operations on scattring matrix

__init__(*N*)

Creator

Parameters

N (*int*) – Dimension of each of the NxN submatrices of the scattring matrix. The total matrix is 2Nx2N

Returns

None.

Methods

S_modes()	Solves the eigenvalue problem of the Bloch modes of the scattring matrix
S_print([i, j])	Function for printing the scattering matrix.
__init__(N)	Creator
add(s)	Recursion method for joining two scattering matrices
add_left(s)	Recursion method for joining two scattering matrices
add_uniform(lay, d)	Recursion method for addig to self the progation matrix of a given layer
add_uniform_left(lay, d)	Recursion method for addig to self the progation matrix of a given layer
der(Sm, Sp[, h])	Calculates the first and second derivative of the scattering matrix with respec to the parameter par.
det()	Calculate the determinat of the scattering matrix
det_modes(kz, d)	
int_complete(S2, u, d)	Retirn the modal coefficient between two scattering matrces (self and S2)
int_f(S2, u)	Retirn the modal coefficient between two scattering matrces (self and S2)
int_f_tot(S2, u, d)	Retirn the modal coefficient between two scattering matrces (self and S2)
left(u1, d1)	Return the "right" inout and output vectors given the "left" ones
matrix()	Returns the full scattering matrix
output(u1, d2)	Returs the output vectors given the input vectors

3.4.2 Methods

S_matrix.add(s)	Recursion method for joining two scattering matrices
S_matrix.add_left(s)	Recursion method for joining two scattering matrices
S_matrix.add_uniform(lay, d)	Recursion method for addig to self the progation matrix of a given layer
S_matrix.add_uniform_left(lay, d)	Recursion method for addig to self the progation matrix of a given layer
S_matrix.der(Sm, Sp[, h])	Calculates the first and second derivative of the scattering matrix with respec to the parameter par.
S_matrix.det()	Calculate the determinat of the scattering matrix
S_matrix.det_modes(kz, d)	
S_matrix.int_f(S2, u)	Retirn the modal coefficient between two scattering matrces (self and S2)
S_matrix.int_f_tot(S2, u, d)	Retirn the modal coefficient between two scattering matrces (self and S2)
S_matrix.left(u1, d1)	Return the "right" inout and output vectors given the "left" ones
S_matrix.matrix()	Returns the full scattering matrix
S_matrix.output(u1, d2)	Returs the output vectors given the input vectors

A_FMM.S_matrix.add

S_matrix.**add**(*s*)

Recursion method for joining two scattering matrices

The connection is between the “right” side of self and the “left” side of s

Parameters
s (S_matrix) – scattering matrix to be joined to self. The

Returns
None.

A_FMM.S_matrix.add_left

S_matrix.**add_left**(*s*)

Recursion method for joining two scattering matrices

The connection is between the “left” side of self and the “right” side of s

Parameters
s (S_matrix) – scattering matrix to be joined to self. The

Returns
None.

A_FMM.S_matrix.add_uniform

`S_matrix.add_uniform(lay, d)`

Recursion method for addig to self the progation matrix of a given layer

The connection is between the "right" side of self and the "left" side of the propagation matrix

Parameters

- **lay** (Layer) – Layer of which to calculate the propagation matrix
- **d** (*float*) – Thickness of the layer

Returns
None.

A_FMM.S_matrix.add_uniform_left

`S_matrix.add_uniform_left(lay, d)`

Recursion method for addig to self the progation matrix of a given layer

The connection is between the "left" side of self and the "right" side of the propagation matrix

Parameters

- **lay** (Layer) – Layer of which to calculate the propagation matrix
- **d** (*float*) – Thickness of the layer

Returns
None.

A_FMM.S_matrix.der

`S_matrix.der(Sm, Sp, h=0.01)`

Calculates the first and second derivative of the scattering matrix with respec to the parameter par.

Parameters

- **Sm** (S_matrix) – S matrix calculated at par=par0-h
- **Sp** (S_matrix) – S matrix calculated at par=par0+h
- **h** (*float, optional*) – Interval used to calculate the derivatives . Defaults to 0.01.

Returns

tuple containing:

- S1 (2darray): First derivative of the scattering matrix with respect to par.
- S2 (2darray): Second derivative of the scattering matrix with respect to par.

Return type
tuple

A_FMM.S_matrix.det

`S_matrix.det()`

Calculate the determinat of the scattering matrix

Returns
Determinant of the scattering matrix

Return type
float

A_FMM.S_matrix.det_modes

`S_matrix.det_modes(kz, d)`

A_FMM.S_matrix.int_f

`S_matrix.int_f(S2, u)`

Retirn the modal coefficient between two scattering matrces (self and S2)

Parameters

- **S2** (S_matrix) – Scattering matrix to between self and the end of the structure
- **u** (*1darray*) – Array of modal coefficient of "left" inputs to self.

Returns

tuple containing:

- uo (TYPE): Array of coefficients of left-propagating modes in the middle
- do (TYPE): Array of coefficients of right-propagating modes in the middle

Return type
tuple

A_FMM.S_matrix.int_f_tot

`S_matrix.int_f_tot(S2, u, d)`

Retirn the modal coefficient between two scattering matrces (self and S2)

Parameters

- **S2** (S_matrix) – Scattering matrix to between self and the end of the structure
- **u** (*1darray*) – Array of modal coefficient of "left" inputs to self.
- **d** (*1darray*) – Array of modal coefficient of "right" inputs to S2

Returns

tuple containing:

- uo (TYPE): Array of coefficients of left-propagating modes in the middle
- do (TYPE): Array of coefficients of right-propagating modes in the middle

Return type
tuple

A_FMM.S_matrix.left

`S_matrix.left(u1, d1)`

Return the "right" inout and output vectors given the "left" ones

Parameters

- **u1** (*1darray*) – Array of modal coefficient of "left" inputs.
- **d1** (*1darray*) – Array of modal coefficient of "left" outputs.

Returns

tuple containing:

- u2 (1darray): Array of modal coefficient of "right" outputs.
- d2 (1darray): Array of modal coefficient of "right" inputs.

Return type
tuple

A_FMM.S_matrix.matrix

`S_matrix.matrix()`

Returns the full scattering matrix

Returns
Scattering matrix as numpy array

Return type
2darray

A_FMM.S_matrix.output

`S_matrix.output(u1, d2)`

Returs the output vectors given the input vectors

Parameters

- **u1** (*1darray*) – Array of modal coefficient of "left" inputs.
- **d2** (*1darray*) – Array of modal coefficient of "right" inputs.

Returns

tuple containing:

- u2 (1darray): Array of modal coefficient of "right" outputs.
- d1 (1darray): Array of modal coefficient of "left" outputs.

Return type
tuple

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

O

P

R

S

T

V

X